
pygame4000

Release 0.0.2

René Dudfield

Dec 14, 2020

CONTENTS:

1	Python game programming tutorial.	1
2	Python game programming part 1 - Introduction	3
2.1	What are these articles going to teach you?	3
2.2	Why python for games?	3
2.3	Python, and python gaming websites.	3
2.4	Installing your python programming environment.	4
2.5	Running the chimp	4
2.6	Next after article introduction	5
3	Python game programming part 2 - Introduction to Python	7
3.1	Variables	7
3.2	Code Blocks	7
3.3	Why Python Code Blocks are Good	8
3.4	Python Is About You	9
3.5	Zen of Python	9
3.6	Python from the Interpreter	10
3.7	Interpreter Conventions	10
3.8	Live, from your living room... It's Python!	11
3.9	Basic Variable Types	11
3.10	Introduction to Container Variable Types	13
3.11	Using Container Variables	14
3.12	Return of the Slice	15
3.13	Run-Time Type Checking	16
3.14	How do you get rid of the trailing space?	17
3.15	What does this code do?	17
3.16	How do you find out what functions you can perform on a dictionary?	17
3.17	How would you find the 5th item from the last in a sequence of 20 items?	17
3.18	Without trying it first, is this legal?	17
3.19	Next	17
4	Python game programming part 3 - Introduction to pygame	19
4.1	Import Modules	19
4.2	Initializing pygame	19
4.3	Setting up the screen	19
4.4	Constructing the monkey filename	20
4.5	Loading the monkey head image	20
4.6	Drawing the monkey onto the screen.	21
4.7	Flipping the display	21
4.8	Adding a way to quit.	21
4.9	The main loop	22
4.10	All the code together.	22
4.11	Exercises	23
5	Python game programming part 4 - More pygame to make your brain hurt	25

5.1	Import Modules	25
5.2	Loading Resources	26
5.3	Game Object Classes	28
5.4	Back on to the Chimp sprite	29
5.5	Initialize Everything	31
5.6	Create The Background	31
5.7	Put Text On The Background, Centered	31
5.8	Display The Background While Setup Finishes	32
5.9	Prepare Game Object	32
5.10	Main Loop	32
5.11	Handle All Input Events	33
5.12	Update the Sprites	33
5.13	Draw The Entire Scene	33
5.14	Game Over	34
5.15	Assignment:	34
6	Python game programming part 5 - Parts of a game	35
6.1	Forgetting the details: Wrapping away implementation details.	35
6.2	Interfacing with the OS	35
6.3	Loading & Drawing graphics	35
6.4	Loading & Playing sound	35
6.5	Reading from Input devices	36
7	Python game programming part 6 - Abstraction	37
7.1	Wrapping up the libraries and forgetting the details	37
7.2	Our sounds module	37
7.3	Implementing our sound module	38
8	Python game programming part 7 - Make a game	43
8.1	A slightly detailed game design.	43
8.2	Sound design	44
8.3	User input design	45
8.4	Code design	46
8.5	Coding the game	46
9	Pixel perfect collision detection in pygame with masks.	51
9.1	Why rectangles aren't good enough.	51
9.2	How is pixel perfect collision detection done? Masks.	52
9.3	How to use pixel perfect collision detection in pygame?	52
9.4	Mask.from_surface with Alpha transparency.	52
9.5	Checking if one mask overlaps another mask.	53
9.6	Pixel perfect collision detection with pygame.sprite classes.	53
9.7	Collision response - approximate collision normal.	53
9.8	Fun uses for masks.	54
10	You know what's awesome? You are!	57
10.1	awesome libraries for pygame	57
11	SDL2 basics tutorial fundamentals in C	59
12	Post modern C tooling	63
12.1	Welcome to the post modern era.	63
12.2	Tools and protection for our feet.	64
12.3	Debuggers	64
12.4	Portable building, and package management	66
12.5	Interpreter and REPL	67
12.6	Testing coverage.	67
12.7	Static analysis	68
12.8	Static analysis tool overview.	69

12.9 Runtime checks and Dynamic Verification	71
12.10 Performance profiling and measurement	74
12.11 Caching builds	75
12.12 Distributed building.	75
12.13 Complexity of code.	75
12.14 Testing your code on different OS/architectures.	76
12.15 Code Formatting	76
12.16 Services	77
12.17 Coding standards for C	77
12.18 How are other projects tested?	77
13 Drag and drop of files with python and pygame	79
14 Finger painting with multi-touch	81
14.1 Finger events	81
14.2 Code: finger_painting_multi_touch.py	82
15 Text Editing Input IME	85
15.1 New events, their fields, and description	85
15.2 Try IME yourself?	86
15.3 Larger example	86
16 Midi musical instrument controllers and synthesizers	91
16.1 What can you do with pygame.midi?	91
16.2 Playing midi files	92
16.3 Listing midi devices with pygame.examples.midi	92
16.4 pygame.midi API basic explainer	92
16.5 Setting up a synth	94
17 Sound generation and drawing	97
17.1 Metronome.	97
18 How to port and market games using #python and #pygame.	99
18.1 a few platforms to port to	99
18.2 Make it a python package	99
18.3 Windows	100
18.4 Flatpak	100
18.5 pypi	100
18.6 Mac	101
18.7 iOS	101
18.8 Steam	101
18.9 Itch.io	102
18.10 Android	102
18.11 Web	102
18.12 Building if you do not have a windows/mac/linux machine	102
18.13 Writing portable python code	103
18.14 Announcing your game.	103
18.15 Icons	103
18.16 Making a game trailer (for youtube)	103
18.17 Animated gif	103
19 Let's write a unit test!	105
19.1 A minimal test	105
19.2 But why write a unit test anyway?	105
19.3 Let's write a unit test!	106
19.4 Standard unittest module.	107
19.5 How to run a single test?	107
19.6 Python 3 to the rescue.	108
19.7 Digression: Good first issue, low hanging fruit, and help wanted.	108

19.8	A full example of a test.	109
19.9	Committing your test, and making a Pull Request.	109
19.10	Writing your pull request text.	110
19.11	What is a git for? Jargon.	111
20	The arduino code	113
21	python pygame code	115
22	Let's make a shit JavaScript Interpreter!	117
23	Let's make a shit javascript interpreter! Part one.	119
23.1	Tokenising	119
24	Let's make a shit javascript interpreter! Part two.	123
24.1	Homework from part One - A simple tokeniser.	123
24.2	Our simple tokeniser	123
24.3	Code for shitjs.	124
24.4	What next? Parsing with the tokens of our simple expression.	124
24.5	What's new is old is new.	124
24.6	Manually stepping through the algorithm.	125
24.7	Exercises for next time	126
24.8	Until next time... Further reading (for the train, or the bathtub).	127
25	Let's explore existing JavaScript implementations.	129
25.1	narcissus - js in js.	129
25.2	Spider monkey.	129
25.3	Google V8.	130
25.4	JSLint.	130
25.5	Rhino	130
25.6	KJS	130
25.7	JavaScriptCore, Squirrelfish, Nitro	131
25.8	Closed source JavaScript implementations	131
25.9	pypy javascript	131
25.10	So what have we learned then?	131
25.11	Exercise for next time	131
25.12	Further reading.	132
26	python best practices	133
26.1	Testing	133
26.2	Documentation	133
26.3	Literate programming	134
26.4	Code quality and formatting	134
26.5	Package your code	134
26.6	Manage dependencies	134
26.7	A private python index	135
27	Indices and tables	137

PYTHON GAME PROGRAMMING TUTORIAL.

Herein lies an introduction to using python for game programming.

Setting up.

- An introduction to what you will learn from this course.
- Setting everything up, and getting started.

Python introduction.

- Introduction to Python.
- Don't be scared. It is python the programing language.

pygame introduction

- Introduction to pygame game programming.
- Now on to the flashy graphics.

More pygame

- More PyGame to make your brain hurt.
- A chimp gets pummeled, and you learn about a basic almost-game.

Parts of a game

- Different parts of a game.
- Loading and displaying graphics, sound, interfacing with OS.

Abstraction

- Wrapping up the library and forgetting the details.
- The makings of a sound module.

A Minimal Game

- minimal game design, sound design, input design, and graphic design.

PYTHON GAME PROGRAMMING PART 1 - INTRODUCTION

2.1 What are these articles going to teach you?

- How to make games with python. Libraries we are going to use include:
 - pygame,
 - numpy

We're going to start with showing you how to set up your python environment. In lecture two we'll move on to showing you the basics of python and pygame. Afterwards we'll build up to making some 2d, and 3d games. Space travel, monkey bashing, ball and block positioning will also be covered.

2.2 Why python for games?

- Python's benefits include clear syntax, fast development, available on multiple platforms, quality free implementations, and open source code. Clear syntax is useful so you can understand what things do, so you can change things quickly. Python allows doesn't require many of the things which other computer languages do. Eg you do not need to type ';' at the end of every line. You do not need brackets for many things. The authors of python try very hard to make sure that python code written remains clear and understandable. Fast development with python is achieved by a few things. No compile times, lower amounts of typing you need to do. You don't need a lengthy compile before you can see your changes. There are even ways where you can change the program whilst it is running! This allows you to tweak your game quicker than compiled languages will allow.

An [open source](#) implementation is pretty important too. The executable too big? Remove parts of python. Recompile it in ways which make your particular game faster. Make all sorts of changes. You don't have to do these things. But if you need to, it's good to have the option.

2.3 Python, and python gaming websites.

- There's a few websites you should explore, to gain true mastery of the python game programming way ;)
 - <https://www.python.org/>
 - <https://www.pygame.org/>

Although we will be teaching python as we go, you may want to check out some other tutorials. If you don't know python already, you should work through one of these other python tutorials over the next two weeks. If python is your first programming language, this is an excellent tutorial to follow:

- <https://docs.python.org/3/tutorial/>

Some gaming websites you should check out include:

- <https://gamedev.net/>

- <https://gamasutra.com/>
- <https://ldjam.com/>

Of particular interest to new games programmers may be the [Glossary of video game terms](#). If you come across any words you don't know there might be the best place to look first for an explanation.

2.4 Installing your python programming environment.

- Time to get into the programming.

<https://www.pygame.org/wiki/GettingStarted>

Many programming books and tutorials have an installation section. Which are almost immediately out of date, and are sometimes not tested.

So the GettingStarted page was started where things are up to date. pygame is backwards compatible. Code written 20 years ago works today.

Don't worry, relax, and check out <https://www.pygame.org/wiki/GettingStarted>

2.4.1 Test your python installation.

- Run python. In windows, go to the start bar find python, and select the IDLE(Python GUI) option. In linux, type python at the command line.

Now you should see a prompt like >>> This is the interactive interpreter, you'll be able to type into here python code, and see it run. Fun. Now type in:

```
import pygame
print(dir(pygame))
print(dir())
```

That should print a bunch of stuff. The import command tells python to load that module. The print command is used to print stuff. duh. The dir command is really cool. It shows you what stuff is inside the pygame module. Now type:

```
help(pygame.Rect)
```

That shows you documentation on a particular object/

[function/ variable](#). Try out help on a few other things inside of the pygame module. eg `help(pygame.sprite)` Check out the documentation for python, and pygame at these places.

- Documentation for pygame: <https://pygame.org/docs/index.html>
- Documentation for python: <https://python.org/docs/>

It's good to know where to look for info when you get stuck, or need to know more details.

2.5 Running the chimp

- From the command prompt:

```
python -m pygame.examples.chimp
```

Now try running some of the other examples.

```
python -m pygame.examples
```

2.6 Next after article introduction

Part Two

PYTHON GAME PROGRAMMING PART 2 - INTRODUCTION TO PYTHON

What is Python?

Python has been accurately described as pseudo-code that runs.

Python is an interpreted language, which means that it is not converted to machine code and is processed by a Python interpreter instead of your computer's CPU. Python compiles into byte-code like Java, unlike Perl. Python uses run-time dynamic typing, unlike Java or C++ which require you to declare types before compiling.

3.1 Variables

Python variables work on references, and have automatic garbage collection so you do not need to worry about allocating or freeing memory. Python variables can be re-assigned at any time to different types of data. For instance, this is perfectly legal:

```
var = 5
var = 5.5
var = 'Five'
```

Some people who are familiar with other languages consider this flexibility to be dangerous, but in practice it is very rarely a problem.

3.2 Code Blocks

Python structures its code based on whitespace (spaces or tabs). This is unusual for programming languages, and is the biggest factor keeping people who are currently programmers in other languages from giving Python a good try. Some existing programmers can never get used to the whitespace code blocks, but the majority of Python converts end up liking whitespace codeblocks even better than the standard bracket method of C++, Java, Perl.

Python accepts new commands on new lines, and accepts code blocks based on the amount of whitespace before any non-whitespace characters. For example:

```
# layer variable used to describe what "layer" of code block indentation we are on
layer = 1
# Always true if statement to show nested code blocks
if True:
    # Nested blocks are indented with 3-4 spaces or a tab character.
    layer = 2
else:
    layer = 2
    if True:
        # Third "layer" of nesting. 2 code blocks deep.
        layer = 3
```

(continues on next page)

(continued from previous page)

```
# Blank lines can be inserted into code blocks, whitespace does not matter
↳ on blank lines
    layer = 3

# After a code block is finished, you revert down to the next code block that
↳ you will continue from.
    # You could drop several could block "layers" at once
    layer = 2
```

Whitespace can consist of spaces or tabs, but it should be consistent through your entire file. Mixing tabs and spaces in a codeblock will cause an error, and mixing them in a file will cause later frustration.

In the example above, you can also see how code blocks are specified. Any line that ends with a colon (:) will tell the Python compiler that the next line should be a sub-code block. This means that the next line should have more whitespace than it's parent line.

It does not matter how many whitespace characters are used, as long as it is more than it's parent and consistent. Usually 3 or 4 spaces are used, or 1 tab. Blank lines are ignored, and it doesn't matter how many whitespace characters are on them.

Comments are done by placing a hash character (#) not in a string. The rest of the line is considered commented.

3.3 Why Python Code Blocks are Good

Why change the standard way to format code? It's been in practice for 40 years, why change now?

Everyone indents their code anyway. Why specify indentation with both whitespace AND brackets? There are two religions to bracket code blocking in C++/Java/Perl, compare with Python:

```
if (something) {
    doStuff();
    while (1) {
        doMore();
    }
}
```

or

```
if (something)
{
    doStuff();
    while (1)
    {
        doMore();
    }
}
```

compared to Python:

```
if something:
    do_stuff()
    while 1:
        do_more()
```

Each of these require you to different eye-gymnastics, and over large file introduce errors. Especially when code block brackets are optional for single line code blocks, making you potentially miss where a code block was not properly specified.

Compare both of the C++/Java/Perl common code bracket methods with the lower Python method of whitespace indentation. Which is easier to read? Which uses the least vertical whitespace? Given 5 pages worth of code, which would you be more likely to make a mistake reading or editing?

Removing unnecessary characters.

Python also promotes removing unnecessary characters from your code. If every line of code is placed on a separate line, why do you need to end every statement with a semi-colon?

Additionally, if parentheses are not necessary, you are not required to have them and English equivalents are available instead of using symbols for logical operations. As a C/C++ programmer for over 15 years, I considered it no big deal to visually parse C symbols, but I noticed after doing things the Python way for a year how much easier it is to read. Compare:

C++/Java:

```
if (a == b && c == d) {
    doStuff();
}
```

Python:

```
if a == b and c == d:
    doStuff()
```

Everyone does it the same way.

No more debates about code style on where to put the brackets, no more having to read differently every file you go to.

Almost.

There is still a choice between spaces and tabs. I'm told spaces are more usual, but I prefer tabs. However, this is very easy to deal with, and most editors can treat them both the same way if you configure them.

However, the differences between peoples code is greatly reduced. Interestingly, Python code usually looks more alike than other languages. Python coders tend to do things the same way, as it is easy, works well, and is encouraged. This is a Good Thing.

3.4 Python Is About You

Everything about Python is created to make development easier for the programmer, as opposed to making programming easier for the compiler or CPU. You have probably heard the often repeated, but often ignored, mantra of "optimize last". Python takes this mantra to heart, and the optimizations of other languages to make their life easier at the expense of you, the programmer, are avoided in Python.

To hear more about reasons why Python has made some smart choices, you can read an interview with Bruce Eckel (author of *Thinking in C++* and *Thinking in Java*) about why he has chosen Python as his language of choice: [Python and the Programmer: A Conversation with Bruce Eckel](#)

3.5 Zen of Python

This sums up Python's goals and aspirations nicely.

```
The Zen of Python (by Tim Peters)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
```

(continues on next page)

(continued from previous page)

```
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

3.5.1 Using Python

Now that you have a little insight into what Python is, you will want to see it in action a little. These lectures are not intended to be full Python tutorials, because there are a lot of good resources out there already for this. What we will do is show you how to use the basics of Python, how to do interesting things with it, how to apply these techniques to game development, and finally how to make games with this knowledge.

3.6 Python from the Interpreter

My first experience with interactively interpreted languages was using Prolog many years ago, and it made a profound impact on me. I always remember Prolog fondly, even though it was very cryptic and hard to do simple things with, because I could watch my results change as I typed them in.

Luckily, interactive interpretation is back, and this time is it now easy to do practical things!

To load up your interpreter run the command “python.exe” on windows:

Start -> Run -> Type: “python”

Or go to the C:Python38\ and double-click “python.exe”.

You should now see a window like this (but taller, I shrunk mine):

interp_01l

If you get an error, or do not see something like this you may have a problem with your installation, or you are clicking in the wrong place. You may need to add the Python directories to your path. You can do this from your Control Panel -> System -> Advanced -> Environment Variables.

To PATH, add C:Python38\ and C:Python38Scripts\

3.7 Interpreter Conventions

In your interpreter the starting characters ‘>>>’ are telling you that it is expecting a new line. The characters ‘...’ is telling you that it expects a continuation of a code block. If you do not properly terminate a command it will show you this ‘...’ prompt until you properly end the command. Check for non-terminated strings or parentheses.

You can press the UP and DOWN keys to cycle through your command history, and LEFT/RIGHT/HOME/END/CTRL+LEFT/CTRL+RIGHT will allow you to move over your line and edit it.

To quit the Python interpreter, press CTRL+Z and then ENTER in Windows or CTRL+D in Unix.

3.8 Live, from your living room... It's Python!

Now that we have the formalities out of the way, it's time to kick things into gear and check out Python in action.

Let's set some variables:

```
>>> i = 5
```

We have now set the variable 'i' to be bound to a object containing the integer 5. If this makes no sense to you, thats ok! Just know that when you assign things like "i = 5", 'i' now is equal to 5. The specifics of what are going on will be explained later, and in most cases, you will never care.

We can inspect the variable by typing it alone:

```
>>> i
5
```

Here we see what the variable now contains. We can get any returned value this way, like so:

```
>>> i * 5
25
```

As long as the value being returned is not being assigned to a variable, then it is output to the interpreter screen. Let's assign it to a variable now:

```
>>> j = i * 5
>>> j
25
```

You can see that after the first operation nothing was printed out, but when we specified 'j' by itself and did not assign it into a variable, then it was output onto the screen. This will help you troubleshoot different functions later on, by letting them fall through into the interpreter so you can see them.

3.9 Basic Variable Types

Let's cover a few different types of data that can be put into a Python variable.

We have already seen that Python takes integers (whole numbers), and it also takes floating point numbers (real numbers), and strings. This makes up the standard set of variables:

```
>>> a = 1
>>> b = 1.0
>>> c = 'One'
>>> a
1
>>> b
1.0
>>> c
'One'
```

Strings can either use single or double quotes, and the only difference between them is which quote you will have to escape. Example:

```
s1 = 'Test "single" quote.'
s2 = "Test 'double' quote."
s3 = 'Test \'single\' escaped quote.'
s4 = "Test \"double\" escaped quote."
```

Everything in Python is an Object. This means that every variable has functions associated with it that can operate on the variable in different ways. You can see the functions by using the dir() function:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__'
↪, '__div__',
'__divmod__', '__doc__', '__float__', '__floordiv__', '__getattr__', '__hash__'
↪, '__hex__',
'__init__', '__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul__'
↪, '__neg__',
'__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__', '__'
↪, '__rand__',
'__rdiv__', '__rdivmod__', '__reduce__', '__repr__', '__rfloordiv__', '__rlshift__'
↪, '__rmod__',
'__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__'
↪, '__rtruediv__',
'__rxor__', '__setattr__', '__str__', '__sub__', '__truediv__', '__xor__']
```

That's a lot of functions inside of a "1" :)

This covers all of the operations that can be done on or against this variable. So "1 * 2" will actually call the function 1.mul(2). The "" characters are used to describe private functions to an object, and while they look a little ugly, you rarely need to use them directly, so it doesn't matter.

You can use the dir() function

Now, let's see some interesting things we can do with strings.

We start out by setting our string variable 's'.

```
>>> s = 'Hi, Im a Python string, and I am all powerful!'
```

Then we run the function upper() on the variable, and we see the result that the string variable has now been returned as upper case. No change is made to the variable 's', because it has not been re-assigned.

```
>>> s.upper()
'HI, IM A PYTHON STRING, AND I AM ALL POWERFUL!'
```

You may be wondering what the dot (.) is in the above command. Remember how we used the dir() function before? Let's try it again:

```
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattr__', '__getitem__', '__getslice__', '__gt__', '__hash__',
'__init__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__repr__', '__rmul__', '__setattr__', '__str__', 'capitalize',
'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join'
↪,
'ljust', 'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip', 'split'
↪,
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
↪, 'zfill']
```

Here are all the commands that are present in the string Object in Python. An object can have functions (sometimes called methods) associated with them. You can call any of these functions by using the above "dot notation", so s.upper() or s.lower() are both available for manipulating this string. In Python, functions that start with two underscores (__) are private, and cannot be called directly.

Next we will search the string for the sub-string 'all'.

```
>>> s.find('all')
33
```

We receive the position 33 for the sub-string 'all', this means that the string 'all' appears at the 33rd position in our string. Let's use that and get an index from our string. This should be familiar to C++ people.

```
>>> s[33]
'a'
```

Now we will try something new that does not exist in the C++ style worlds. We will use what is called a *slice* in Python. This can be used on anything that is a *sequence*, which strings are. We will cover other sequences shortly.

```
>>> s[33:] 'all powerful!'
```

We have gotten the slice “from position 33 to the end of the string” with the above command. Now let’s try the opposite.

```
>>> s[:33] 'Hi, Im a Python string, and I am '
```

Here we have from the beginning of the string to position 33. Slicing is very powerful, and there are several additional ways to use slicing that we will cover later.

3.10 Introduction to Container Variable Types

Up to now we have seen some pretty standard programming types, but containers are where Python shines. If for no other reason, this feature set was the one that really enamored me to Python. The simplicity and elegance of Python containers is quite something, in my humble opinion.

Now that I have unreasonably (but correctly :)) set expectations, let’s see what the containers are:

Sequence:

```
>>> a = (1, 3, 5, 7)
>>> a
(1, 3, 5, 7)
>>> a[1]
3
```

A sequence is a series of variables that are contained together in... sequence. They can be any variable types, including other sequence or containers. Sequences are immutable, which means that you cannot add, remove or change items once the sequence is created. This is very similar to an Array in Java, and has some similarities to arrays in C, except that you cannot change data in the elements.

List:

```
>>> b = [2, 4, 6, 8]
>>> b
[2, 4, 6, 8]
>>> b[1]
4
```

A list is sequence that is mutable. You can add, remove and change elements in it. Lists have methods (another name for a function inside an Object) for sorting and reversing the list. Lists can be treated as stacks or queues with `append()`, `pop()` and `remove()` functions.

Dictionary:

```
>>> c = {'a':0, 'b':1, 'c':2}
>>> c
{'a': 0, 'c': 2, 'b': 1}
>>> c['b']
1
```

A dictionary is Python’s version of a hash or map. It assigns a value to a key, as is standard for “key-pair” stored data. This gives immediate access to any data, so that you do not have to step through an entire list to find something, provided you know it’s key.

If you were wondering. While you can do:

```
>>> c['b']
1
```

You cannot do:

```
>>> c[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 1
```

Because dictionaries are not 2-way hashes. This isn't to say that this can't be done, and in fact it is not very difficult at all to do, but it is not default functionality. By the way, using integers for dictionary keys is perfectly acceptable, this example failed simply because there was no key of 1.

3.11 Using Container Variables

Feeling underwhelmed by the Amazing Python Containers? Hopefully not too much, but at first look they are not so much as amazing as simple “readable” in my book. Their real power comes from Python's ability to mix types together. For instance, this is valid:

```
>>> d = [5, 5.5, 'Five to Five', (5, 6)]
>>> d
[5, 5.5, 'Five to Five', (5, 6)]
>>> d[1]
5.5
>>> d[2]
'Five to Five'
```

I have now mixed 4 different types of data into a single list. An integer, a float, a string and a sequence are all living happily together. What's more, they have all retained their normal state and did not have to be turned into “generic objects” as they would have to be in some other Object Oriented languages, so they can be extracted and used as-is.

I believe in Data Driven Programming, so while I have named this section to be able containers I am going to introduce you to various other Python concepts for handling sequences. Let's see how we can use some of this container goodness:

```
>>> for item in d:
...     print(item)
...
5
5.5
Five to Five
(5, 6)
```

Introducing Python's version of the for loop. Python breaks tradition once again, and instead of setting a starting variable, checking that it is still in range, and incrementing, it simply iterates over a sequence.

'd' is a list, which is a mutable sequence. This sequence is then cycled over, starting at the first item and assigning it into the variable 'item' and executing the code block. After the code block, the for loop returns to the 'd' variable and retrieves the next 'item' in the sequence. When it is out of items, it moves past the for loop. Elegant.

Let's see it work over our dictionary.

```
>>> c
{'a': 0, 'c': 2, 'b': 1}
>>> c.keys()
['a', 'c', 'b']
```

(continues on next page)

(continued from previous page)

```
>>> for key in c.keys():
...     print('%s: %d' % (key, c[key]))
...
a: 0
c: 2
b: 1
```

We start by checking the contents of 'c' again. A dictionary with string keys attached to integer values.

We want to loop over the keys in this dictionary, so let's look at the keys with the function keys(). Here we can see all the keys in the dictionary returned in a list.

So now we will do a for loop over the 'c.keys()' returned list, and the variable key will contain the key for each dictionary entry. We want to print out the results, so we will use the print() command, again because Python likes to conserve the use of symbols you can see that I do not need to type the parentheses around print (but I could if I wanted to).

If you are familiar with sprintf() in C++, you may find this a pleasant surprise. Any string can be formatted by the syntax sugar of: 'string %s ' % variable.

Here we are substituting a string (%s) and a integer (%d), and so we feed in a Sequence of the key and 'c' dictionary value at index 'key' into the format command. This new formatted string is then passed to the print() function and output to the interpreter.

If there was only one variable, you would not need the parentheses to build a Sequence.

3.12 Return of the Slice

Remember slicing strings? Well, I said strings were sequences and any sequences can be sliced, so let's try it out on a list:

```
>>> b = [2, 4, 6, 8, 10, 12, 14, 16]
>>> b
[2, 4, 6, 8, 10, 12, 14, 16]
```

We start off with a basic list of even numbers.

```
>>> b[4]
10
```

We can get an index out of the list, this is like a 1 item slice.

```
>>> b[4:]
[10, 12, 14, 16]
```

Here we slice from position 4 to the end of the sequence.

```
>>> b[:4]
[2, 4, 6, 8]
```

And here we slice from the beginning of the sequence to the 5th position. Something worth mentioning: computers do not start counting from 1, they start from 0. So position 4, is actually the 5th position. 0, 1, 2, 3, 4.

```
>>> b[4:6]
[10, 12]
```

Here we slice from the 5th position to the before 7th position. The end position is not returned, only up to it.

```
>>> b[-1]
16
```

Here we use a negative slice to select the last item in the sequence.

```
>>> b[1:-1]
[4, 6, 8, 10, 12, 14]
```

Here we select from the 2nd position to the 2nd to last position. Again, the end position is not returned, since -1 gives us the “last element”, then doing a slice to -1 gives us up to the “2nd to last” element.

As you can see, slicing can give you powerful access to sequences.

3.13 Run-Time Type Checking

Python may not require you to specify variable types in your code, but that does not mean Python does not care what your variable types are. Mixing variable types is only allowed in circumstances where it is seen as a proper thing to do, in all other situations you must first convert the variables to a common type before they can be used together.

Integers and Floats are converted into Floats.

```
>>> 5 + 5.0
10.0
```

Lists may be added together. Sequences may be added together.

```
>>> [1,3] + [3,4]
[1, 3, 3, 4]
>>> (1,2) + (3,4)
(1, 2, 3, 4)
```

Lists and sequences must not be added together. Convert the sequence into a list first.

```
>>> (1,2) + [3,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can only concatenate tuple (not "list") to tuple
>>>
>>> list((1,2)) + [3,4]
[1, 2, 3, 4]
```

Lists and sequences may be multiplied by integers. They may not have other operations done on them with integers.

```
>>> (1,2) * 2
(1, 2, 1, 2)
>>>
>>> (1,2) + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can only concatenate tuple (not "int") to tuple
```

Strings, as sequences, can also be multiplied by integers.

```
>>> 'PYTHON! ' * 3
'PYTHON! PYTHON! PYTHON! '
```

When formatting strings, converting from an integer to a string is allowed. The opposite is not true.

```
>>> 'Str: %s Num: %s' % ('python', 5)
'Str: python Num: 5'
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 'Str: %s Num: %d' % ('python', 'Five')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required
```

So if you refer back to the section containing the Zen of Python, you will see it's explanations in effect here. Where things are expected, they are performed. Where they are not expected, errors are thrown.

3.13.1 Exercises

3.14 How do you get rid of the trailing space?

```
>>> 'PYTHON! ' * 3
'PYTHON! PYTHON! PYTHON! '
```

3.15 What does this code do?

```
a = {15:'dog', 22:'cat', 35:'beard', 99:'emu', 101:'llama'}
b = (38, 22, 19, 99, 22, 14, 100, 101, 15)
for item in b:
    if a.has_key(item):
        a[a[item]] = item
```

3.16 How do you find out what functions you can perform on a dictionary?

3.17 How would you find the 5th item from the last in a sequence of 20 items?

3.18 Without trying it first, is this legal?

```
>>> (1,2) + [4,5]
```

3.19 Next

Part Three

PYTHON GAME PROGRAMMING PART 3 - INTRODUCTION TO PYGAME

Now onto the flashy graphics!

We're going to show you pygame now. Going line by line through a simple example which puts a monkey head onto a window.

If you want to see the full code listing skip to the bottom of the lecture. For now we'll start exploring the first few lines.

4.1 Import Modules

Below is the code that imports all the needed modules into your program. [Modules](#) are a grouping of code, or a library.

Open a command prompt and go into the pygame examples directory. Then run python.

```
>>> import pygame, sys, os
>>> from pygame.locals import *
```

Try typing that into your interpreter like this: **!import_stuff!**

You can look up the documentation for these modules: [pygame](#), [os](#), [sys](#).

You may also want to try using `dir()` and `help()` functions on those modules :)

```
>>> dir(pygame)
```

4.2 Initializing pygame

Now we initialize all imported pygame modules. This is done with the `pygame.init()` function.

```
>>> pygame.init()
(6, 0)
```

We've now initialized the video, the sound, and a number of other pygame modules. If you type it into the interpreter you'll see that 6 modules have initialized correctly, and none have failed.

4.3 Setting up the screen

First we are making a window 468 pixels wide, and 60 pixels high. A wide, short window.

```
window = pygame.display.set_mode((468, 60))
```

Check out the documentation for `pygame.display.set_mode`,

Now we set a caption on the window. The caption is the text in the middle of the window bar. Set the text to what ever you like! Change ‘Monkey Fever’ to ‘I am the best python coder in the room!’.

```
pygame.display.set_caption('Monkey Fever')
```

Check out the documentation for `pygame.display.set_caption`,

Finally we get the display surface representing a screen.

```
screen = pygame.display.get_surface()
```

A surface represents either the screen or an area in memory where images are held.

Check out the documentation for `pygame.Surface`,

4.4 Constructing the monkey filename

```
>>> monkey_head_file_name = os.path.join("data", "chimp.bmp")
>>> print(monkey_head_file_name)
data\chimp.bmp
```

This creates a relative pathname to the file. In this example all the resources are in a “data” subdirectory of the pygame examples directory. By using the `os.path.join` function, a pathname will be created that works for whatever platform the game is running on.

To open the pygame documentation you can run this command.

```
python -m pygame.docs
```

To find the folder where the examples are type this command into the interpreter.

```
>>> import pygame.examples
>>> print(pygame.examples.__file__)
```

The relative path “datachimp.bmp” is relative to the examples directory. The full path name would be

```
"C:\\Program Files\\Pygame-Docs\\examples\\data\\chimp.bmp"
```

On debian linux the pygame examples directory is in `/usr/share/doc/python3.7-pygame/examples/` The relative path “data/chimp.bmp” is relative to that examples directory. The full path name would be

```
"/usr/share/doc/python3.7-pygame/examples/data/chimp.bmp"
```

Hopefully you can see that `os.path.join()` is one of the python functions which helps you write code which will run on multiple platforms. The same code should run on windows, linux boxes, macs, and other types of machines.

4.5 Loading the monkey head image

Now we get to load the monkey head image.

```
>>> monkey_surface = pygame.image.load(monkey_head_file_name)
```

simple eh?

Have a look at the documentation for the `pygame.image.load` function.

4.6 Drawing the monkey onto the screen.

```
screen.blit(monkey_surface, (0,0))
```

Here we come accross the blit function. *blit* is just another word for draw.

What we are doing on this line is drawing the image of the monkey, which is contained in the `monkey_surface` variable, onto the screen.

The second argument to the function is telling *blit* to draw the monkey at coordinates (0,0). The top left of the screen.

Pygame uses a coordinate system for the screen where `x= 0, y=0` is the top left of the screen. `(0,20)` is twenty pixels below the top of the screen.

4.7 Flipping the display

```
pygame.display.flip()
```

Here is where we flip the display surface. This updates the whole screen.

There are more complicated things which you can do with updating the display, which we will explain in the upcoming lectures. If you want to learn more about updating the display you can find out here - <https://www.pygame.org/docs/ref/display.html#pygame.display.flip>

Why do you have to flip the display? To see your graphics drawn. Flipping the display is your way of telling pygame that you have finished making changes for that frame, now please show the changes.

For now just be content that you should flip the display after having drawn to it.

4.8 Adding a way to quit.

```
def input(events):
    for event in events:
        if event.type == QUIT:
            sys.exit(0)
        else:
            print(event)
```

We are defining a function with this code. This function does two things:

- looks for a quit event.
- prints other events.

An event is:

- Something that takes place; an occurrence.
- A significant occurrence or happening.
- A social gathering or activity.

Our game isn't likely to get an invitation down to the pub, or the park, but it may be told that the mouse has moved, that certain keys have been pressed or the joystick has been moved.

These are the types of events that happen within our program.

The `input()` function above loops over the input sequence *events*, and does a test on each event.

Once a quit event happens the program exits. A quit event can happen by clicking on the close window, or pressing ALT+F4.

If it is not a quit event it prints the event to the console(command line window).

4.9 The main loop

```
while True:
    input(pygame.event.get())
```

Here we have an infinite loop. While True is true it will keep looping. As true is going to stay true for a long time, it will keep going on(probably until the program exits).

`pygame.event.get` is used to see what is happening in the program. It returns a list of events. We pass this list to the input function we defined above.

4.10 All the code together.

Below we have all the code together. Copy this into your text editor and save it in the Pygame-Docsexamples directory as `monkey_fever.py`

Then run it, and see all the events fly by on the console!

```
import pygame, sys, os
from pygame.locals import *

pygame.init()

window = pygame.display.set_mode((468, 60))
pygame.display.set_caption('Monkey Fever')
screen = pygame.display.get_surface()

monkey_head_file_name = os.path.join("data", "chimp.bmp")

monkey_surface = pygame.image.load(monkey_head_file_name)

screen.blit(monkey_surface, (0,0))
pygame.display.flip()

def input(events):
    for event in events:
        if event.type == QUIT:
            sys.exit(0)
        else:
            print(event)

while True:
    input(pygame.event.get())
```

!monkey_fever!

As you can see the monkey looks a bit strange. It's got red in the background. The red color is what is called a color key. That is a color in the image which represents transparency. So instead of showing red it should show nothing, and let the black background be seen.

Now try pressing a few keys. You will notice the events being printed out to the console. If you press a mouse button you will get events for that. If you move the mouse you will get events.

Read the pygame docs for events, they are quite good - <https://www.pygame.org/docs/ref/event.html>

Also try out this example

```
python -m pygame.examples.eventlist
```

4.11 Exercises

4.11.1 Move the head

- Make the monkeys head be drawn 35 pixels to the right. Then make it drawn 40 pixels from the top of the window.

4.11.2 Quit on any key pressed

- Find out how to make the program quit when you press any key. Once you find out, make your program quit when any key is pressed.

4.11.3 Find the size of the monkey surface

- Print to the console the size of the monkey surface.

4.11.4 Move the head when pressing a key

- When pressing the 's' key make the monkey move to x = 0 and y = 0. Make it move to x = 35 y = 40 when the 'd' key is pressed.

4.11.5 Read pygame examples

- Read through some of the pygame examples. Run them see what they do. You probably won't understand them all, but you will likely get a feel for some other pygame code. For those new to python you should also have a read through some of the python tutorials:

- <https://docs.python.org/3/tutorial/>
- <https://python.org/doc/Intros.html>

4.11.6 Next

In the next article I will be showing you:

- how to load and play sounds,
- some more advanced stuff with updating the display,
- how to load color keys in images properly.
- an introduction to the pygame sprite class.

Until next time. Have fun!

Part Four

PYTHON GAME PROGRAMMING PART 4 - MORE PYGAME TO MAKE YOUR BRAIN HURT

From last weeks pygame we are now going to move on to a more featureful demonstration. This lecture is based on the Chimp line by line tutorial found at the pygame site. Thanks to Pete Shinnars for letting us base this lecture on his tutorial.

In the *pygame* examples there is a simple example named, “chimp”. This example simulates a punchable monkey moving around a small screen with promises of riches and reward. The example itself is very simple, and a bit thin on errorchecking code. This example program demonstrates many of pygame’s abilities, like creating a graphics window, loading images and sound files, rendering TTF text, and basic event and mouse handling.

The program and images can be found inside the standard source distribution of pygame. To find the examples use

```
python -c "import pygame.examples;print(pygame.examples.__file__)"
```

Alternatively you can find the examples by typing this into the interpreter REPL.

```
>>> import pygame.examples
>>> print(pygame.examples.__file__)
```

With your text editor open up chimp.py in that directory.

You should have been able to run chimp.py from the command line. If not, go back to [Part 1 introduction](#) and figure out how. As we go on you’ll be changing a few things in the chimp.py, so you may want to make a backup copy.

This tutorial will go through the code block by block. Explaining how the code works. There will also be mention of how the code could be improved and what errorchecking could help out.



5.1 Import Modules

This is the code that imports all the needed modules into your program. It also checks for the availability of some of the optional pygame modules. Modules <https://docs.python.org/3/tutorial/modules.html> ‘__’ are a grouping of code, or a library.

```
import os, sys
import pygame
from pygame.locals import *

if not pygame.font: print('Warning, fonts disabled')
if not pygame.mixer: print('Warning, sound disabled')
```

In the next line, we import the pygame package. When pygame is imported it imports all the modules belonging to pygame. Some pygame modules are optional, and if they aren't found, their value is set to "None". None is like NULL in other programming languages.

There is a special *pygame* module named "locals". The members of this module are commonly used constants and functions that have proven useful to put into your program's global namespace. This locals module includes functions like "Rect" to create a rectangle object, and many constants like "QUIT, HWSURFACE" that are used to interact with the rest of *pygame*. Importing the locals module into the global namespace like this is entirely optional. If you choose not to import it, all the members of locals are always available in the *pygame* module.

Lastly, we decide to print a nice warning message if the font or sound modules in pygame are not available.

5.2 Loading Resources

Now we are going to expand on what we did in the last lecture with loading images. We are also going to be adding a sound loading function.

We're going to add some error checking, and set the color key for the image. We put this code in a function so that we can reuse it to load multiple images without having to type all the code in every time we want to load an image.

This code is from *chimp.py* which should be open in your text editor.

```
def load_image(name, colorkey=None):
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
    except pygame.error, message:
        print("Cannot load image:", name)
        raise SystemExit, message
    image = image.convert()
    if colorkey is not None:
        if colorkey is -1:
            colorkey = image.get_at((0,0))
        image.set_colorkey(colorkey, RLEACCEL)
    return image, image.get_rect()
```

5.2.1 Some new python concepts

For those of you new to python, we are now going to describe some of the concepts introduced here. Don't worry if you don't get them all, you'll probably need to read this section a few times. I have made links to websites that describe each of the concepts in more detail. You should probably go and read those sections as well.

```
def load_image(name, colorkey=None):
    fullname = os.path.join('data', name)
```

First two lines of the *load_image* function/

def is used to tell python that you are starting a new function. A function is a piece of code which does something. It means you don't need to retype code all the time. Instead of typing all that stuff out you can just call:

```
im = load_image("chimp.bmp")
```

Further on in the code you see the *try*, and *except* being used. This is for error handling. You try to load the image, and if there is an unexpected error the code in the *except* block gets called. Read up more about [errors and exceptions](#).

5.2.2 Loading images explained

The `load_image` function takes the name of an image to load. It also optionally takes an argument it can use to set a colorkey for the image. A colorkey is used in graphics to represent a color of the image that is transparent.

The first thing this function does is create a full pathname to the file. In this example all the resources are in a “data” subdirectory. By using the `os.path.join` function, a pathname will be created that works for whatever platform the game is running on.

Remember that we can look at the documentation for functions in the interactive interpreter. In the interactive interpreter type:

```
import os.path
help(os.path.join)
```

Next we load the image using the `pygame.image.load` function. We wrap this function in a try/except block, so if there is a problem loading the image, we can exit gracefully. After the image is loaded, we make an important call to the `convert()` function. This makes a new copy of a Surface and converts its color format and depth to match the display. This means blitting the image to the screen will happen as quickly as possible.

Images can be in many different color formats. For example RGB with 8 bits for red, green and blue. Or 8 bit indexed color, or RGBA (Red,Green,Blue,Alpha). The more bits used for each pixel on an image, the more colors it can show. For an explanation of surfaces check out <https://www.pygame.org/docs/ref/surface.html>

Last, we set the colorkey for the image. If the user supplied an argument for the colorkey argument we use that value as the colorkey for the image. This would usually just be a color RGB value, like (255, 255, 255) for white. You can also pass a value of -1 as the colorkey. In this case the function will lookup the color at the topleft pixel of the image, and use that color for the colorkey.

5.2.3 Loading sound explained

```
def load_sound(name):
    class NoneSound:
        def play(self): pass
    if not pygame.mixer:
        return NoneSound()
    fullname = os.path.join('data', name)
    try:
        sound = pygame.mixer.Sound(fullname)
    except pygame.error, message:
        print("Cannot load sound:", fullname)
        raise SystemExit, message
    return sound
```

Next is the function to load a sound file. The first thing this function does is check to see if the `pygame.mixer` module was imported correctly. If not, it returns a small class instance that has a dummy play method. This will act enough like a normal `Sound` object for this game to run without any extra error checking.

If you’re wondering what a class is read up on them at these places:

- <https://python.org/doc/current/tut/node11.html>,
- https://diveintopython.org/fileinfo_divein.html,
- <http://ibiblio.org/obp/thinkCS/python/english/chap12.htm>.

You will need to know about classes for the sections below, where we make and describe the `Fist`, and `Chimp` classes.

This function is similar to the image loading function, but handles some different problems. First we create a full path to the sound image, and load the sound file inside a try/except block. Then we simply return the loaded `Sound` object.

Pygame can load a number of different sound files.

- .ogg files. A free high quality lossy sound format. <http://www.vorbis.com/>
- .mp3 files. A popular lossy format. If you haven't heard of these, pull up a random teenager and ask them about it.
- .wav files. Of various types. These are usually uncompressed sound formats.

It can also open up mod files, and midi files for music.

5.3 Game Object Classes

Here we create two classes to represent the objects in our game. Almost all the logic for the game goes into these two classes. We will look over them one at a time here.

```
class Fist(pygame.sprite.Sprite):
    """moves a clenched fist on the screen, following the mouse"""
    def __init__(self):
        pygame.sprite.Sprite.__init__(self) #call Sprite initializer
        self.image, self.rect = load_image(fist.bmp, -1)
        self.punching = 0

    def update(self):
        "move the fist based on the mouse position"
        pos = pygame.mouse.get_pos()
        self.rect.midtop = pos
        if self.punching:
            self.rect.move_ip(5, 10)

    def punch(self, target):
        "returns true if the fist collides with the target"
        if not self.punching:
            self.punching = 1
            hitbox = self.rect.inflate(-5, -5)
            return hitbox.colliderect(target.rect)

    def unpunch(self):
        "called to pull the fist back"
        self.punching = 0
```

The players fist is represented by the class above.

It is derived from the Sprite class included in the `pygame.sprite` module. The `init` function is called when new instances of this class are created. The first thing we do is be sure to call the `init` function for our base class. This allows the Sprite's `init` function to prepare our object for use as a sprite. This game uses one of the sprite drawing Group classes. These classes can draw sprites that have an "image" and "rect" attribute. By simply changing these two attributes, the renderer will draw the current image at the current position.

Unlike in the previous lecture where we *blit* the image directly to the screen, here we put the images in sprite classes. This gives us some advantages. Mainly the sprite classes are used for organising drawing of images. We want to draw as little as possible, which the sprite classes can do for us by keeping track of where the images we draw go. If you want to know more about sprites read <https://www.pygame.org/docs/tut/SpriteIntro.html>.

All sprites have an `update()` method. This function is typically called once per frame. It is where you should put code that moves and updates the variables for the sprite. The `update()` method for the fist moves the fist to the location of the mouse pointer. It also offsets the fist position slightly if the fist is in the "punching" state.

The `punch()` and `unpunch()` methods change the punching state for the fist. The `punch()` method also returns a true value if the fist is colliding with the given target sprite.

5.3.1 Don't be a square; detour into the world of Rect

Ok a short detour from the Sprite classes to describe Rects. `Rect` objects are simply classes which represent a rectangle. However they are very featureful.

They are used throughout pygame to help you organise and optimize drawing images. They can be used for collision detection and moving images accross the screen. You can check to see if a point is within a rectangle. There are many things you can do with them. Just check out the documentation.

```
my_rect = pygame.Rect(20, 25, 40, 50)
```

That makes a Rect object with its top corner at x= 20 y = 25. It has a width of 40 pixels and a height of 50 pixels.

You can see that in the `Fist.punch()` method it uses a `collidirect()` call to see if the target(usually the rect for our unfortunate chimp) has collided with the fists rect attribute. It uses a slightly smaller rect than the fists rect, so that it is slightly harder to punch the chimp. It uses the `Rect.inflate` method to make a smaller rect.

5.3.2 Python note on docstrings

In this class we see docstrings being used. Docstrings are documentation or comments used to describe what functions do. Docstrings are not like normal comments, in that they are used to generate online documentation. When you do a:

```
>>> def x():
...     """prints the letter x"""
...     print("x")
...
>>> help(x)

Help on function x in module __main__:

x()
    prints the letter x
>>> print(x.__doc__)
prints the letter x
>>>
```

In this function the `"""prints the letter x"""` is a doc string. A doc string is the line immediately after the start of a class or function declaration.

You can also access the docstring through the doc attribute.

5.4 Back on to the Chimp sprite

Ok, now we are going to explain the Chimp class.

```
class Chimp(pygame.sprite.Sprite):
    """moves a monkey critter across the screen. it can spin the
    monkey when it is punched."""
    def __init__(self):
        pygame.sprite.Sprite.__init__(self) #call Sprite intializer
        self.image, self.rect = load_image('chimp.bmp', -1)
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.rect.topleft = 10, 10
        self.move = 9
        self.dizzy = 0

    def update(self):
```

(continues on next page)

```

        "walk or spin, depending on the monkeys state"
        if self.dizzy:
            self._spin()
        else:
            self._walk()

    def _walk(self):
        "move the monkey across the screen, and turn at the ends"
        newpos = self.rect.move((self.move, 0))
        if self.rect.left < self.area.left or \
            self.rect.right > self.area.right:
            self.move = -self.move
            newpos = self.rect.move((self.move, 0))
            self.image = pygame.transform.flip(self.image, 1, 0)
        self.rect = newpos

    def _spin(self):
        "spin the monkey image"
        center = self.rect.center
        self.dizzy = self.dizzy + 12
        if self.dizzy >= 360:
            self.dizzy = 0
            self.image = self.original
        else:
            rotate = pygame.transform.rotate
            self.image = rotate(self.original, self.dizzy)
        self.rect = self.image.get_rect()
        self.rect.center = center

    def punched(self):
        "this will cause the monkey to start spinning"
        if not self.dizzy:
            self.dizzy = 1
            self.original = self.image

```

The chimp class is doing a little more work than the fist, but nothing more complex. This class will move the chimp back and forth across the screen. When the monkey is punched, he will spin around to exciting effect. This class is also derived from the base Sprite class, and is initialized the same as the fist. While initializing, the class also sets the attribute “area” to be the size of the display screen.

The update function for the chimp simply looks at the current “dizzy” state, which is true when the monkey is spinning from a punch. It calls either the `_spin` or `_walk` method. These functions are prefixed with an underscore. This is just a standard python idiom which suggests these methods should only be used by the Chimp class. We could go so far as to give them a double underscore, which would tell python to really try to make them private methods, but we don’t need such protection. :)

The `_walk` method creates a new position for the monkey by moving the current rect by a given offset. If this new position crosses outside the display area of the screen, it reverses the movement offset. It also mirrors the image using the `pygame.transform.flip` function. This is a crude effect that makes the monkey look like he’s turning the direction he is moving.

The `_spin` method is called when the monkey is currently “dizzy”. The dizzy attribute is used to store the current amount of rotation. When the monkey has rotated all the way around (360 degrees) it resets the monkey image back to the original unrotated version. Before calling the `transform.rotate` function, you’ll see the code makes a local reference to the function simply named “rotate”. There is no need to do that for this example, it is just done here to keep the following line’s length a little shorter.

Note that when calling the rotate function, we are always rotating from the original monkey image. When rotating, there is a slight loss of quality. Repeatedly rotating the same image and the quality would get worse each time.

Also, when rotating an image, the size of the image will actually change. This is because the corners of the image will be rotated out, making the image bigger. We make sure the center of the new image matches the center of the

old image, so it rotates without moving.

The last method is `punched()` which tells the sprite to enter its dizzy state. This will cause the image to start spinning. It also makes a copy of the current image named “original”.

5.5 Initialize Everything

Before we can do much with `pygame`, we need to make sure its modules are initialized. In this case we will also open a simple graphics window. Now we are in the `main()` function of the program, which actually runs everything.

```
pygame.init()
screen = pygame.display.set_mode((468, 60))
pygame.display.set_caption(Monkey Fever)
pygame.mouse.set_visible(0)
```

The first line to initialize `pygame` takes care of a bit of work for us. It checks through the imported `pygame` modules and attempts to initialize each one of them. It is possible to go back and check if modules failed to initialize, but we won't bother here. It is also possible to take a lot more control and initialize each specific module by hand. That type of control is generally not needed, but is available if you desire.

Next we set up the display graphics mode. Note that the `pygame.display` module is used to control all the display settings. In this case we are asking for a simple skinny window. There is an entire separate tutorial on setting up the graphics mode, but if we really don't care, `pygame` will do a good job of getting us something that works. Pygame will pick the best color depth, since we haven't provided one.

Last we set the window title and turn off the mouse cursor for our window. Very basic to do, and now we have a small black window ready to do our bidding. Usually the cursor defaults to visible, so there is no need to really set the state unless we want to hide it.

5.6 Create The Background

Our program is going to have text message in the background. It would be nice for us to create a single surface to represent the background and repeatedly use that. The first step is to create the surface.

```
background = pygame.Surface(screen.get_size())
background = background.convert()
background.fill((250, 250, 250))
```

This creates a new surface for us that is the same size as the display window. Note the extra call to `convert()` after creating the Surface. The convert with no arguments will make sure our background is the same format as the display window, which will give us the fastest results.

We also fill the entire background with a solid white-ish color. Fill takes an RGB triplet as the color argument.

5.7 Put Text On The Background, Centered

Now that we have a background surface, let's get the text rendered to it. We only do this if we see the `pygame.font` module has imported properly. If not, we just skip this section.

```
if pygame.font:
    font = pygame.font.Font(None, 36)
    text = font.render("Pummel The Chimp, And Win $$$", 1, (10, 10, 10))
    textpos = text.get_rect()
    textpos.centerx = background.get_rect().centerx
    background.blit(text, textpos)
```

As you see, there are a couple steps to getting this done. First we must create the font object and render it into a new surface. We then find the center of that new surface and blit (paste) it onto the background.

The font is created with the font module's `Font()` constructor. Usually you will pass the name of a truetype font file to this function, but we can also pass `None`, which will use a default font. The `Font` constructor also needs to know the size of font we want to create.

We then render that font into a new surface. The render function creates a new surface that is the appropriate size for our text. In this case we are also telling render to create antialiased text (for a nice smooth look) and to use a dark grey color.

Next we need to find the centered position of the text on our display. We create a “Rect” object from the text dimensions, which allows us to easily assign it to the screen center.

Finally we blit (blit is like a copy or paste) the text onto the background image.

5.8 Display The Background While Setup Finishes

We still have a black window on the screen. Lets show our background while we wait for the other resources to load.

```
screen.blit(background, (0, 0))
pygame.display.flip()
```

This will blit our entire background onto the display window. The blit is self explanatory, but what about this flip routine?

In pygame, changes to the display surface are not immediately visible. Normally, a display must be updated in areas that have changed for them to be visible to the user. With double buffered displays the display must be swapped (or flipped) for the changes to become visible. In this case the `flip()` function works nicely because it simply handles the entire window area and handles both singlebuffered and doublebufferes surfaces.

5.9 Prepare Game Object

Here we create all the objects that the game is going to need.

```
whiff_sound = load_sound('whiff.wav')
punch_sound = load_sound('punch.wav')
chimp = Chimp()
fist = Fist()
allsprites = pygame.sprite.RenderPlain((fist, chimp))
clock = pygame.time.Clock()
```

First we load two sound effects using the `load_sound` function we defined above. Then we create an instance of each of our sprite classes. And lastly we create a sprite Group which will contain all our sprites.

We actually use a special sprite group named `RenderPlain`. This sprite group can draw all the sprites it contains to the screen. It is called `RenderPlain` because there are actually more advanced Render groups. But for our game, we just need simple drawing. We create the group named “allsprites” by passing a list with all the sprites that should belong in the group. We could later on add or remove sprites from this group, but in this game we won't need to.

The clock object we create will be used to help control our game's framerate. We will use it in the main loop of our game to make sure it doesn't run too fast.

5.10 Main Loop

Nothing much here, just an infinite loop.

```
while 1:
    clock.tick(60)
```

All games run in some sort of loop. The usual order of things is to check on the state of the computer and user input, move and update the state of all the objects, and then draw them to the screen. You'll see that this example is no different.

We also make a call to our clock object, which will make sure our game doesn't run faster than 60 frames per second.

5.11 Handle All Input Events

This is an extremely simple case of working the event queue.

```
for event in pygame.event.get():
    if event.type == QUIT:
        return
    elif event.type == KEYDOWN and event.key == K_ESCAPE:
        return
    elif event.type == MOUSEBUTTONDOWN:
        if fist.punch(chimp):
            punch_sound.play() #punch
            chimp.punched()
        else:
            whiff_sound.play() #miss
    elif event.type == MOUSEBUTTONUP:
        fist.unpunch()
```

First we get all the available Events from pygame and loop through each of them. The first two tests see if the user has quit our game, or pressed the escape key. In these cases we just return from the main() function and the program cleanly ends.

Next we just check to see if the mouse button was pressed or released. If the button was pressed, we ask the fist object if it has collided with the chimp. We play the appropriate sound effect, and if the monkey was hit, we tell him to start spinning (by calling his punched() method).

5.12 Update the Sprites

```
allsprites.update()
```

Sprite groups have an update() method, which simply calls the update method for all the sprites it contains. Each of the objects will move around, depending on which state they are in. This is where the chimp will move one step side to side, or spin a little farther if he was recently punched.

5.13 Draw The Entire Scene

Now that all the objects are in the right place, time to draw them.

```
screen.blit(background, (0, 0))
allsprites.draw(screen)
pygame.display.flip()
```

The first blit call will draw the background onto the entire screen. This erases everything we saw from the previous frame (slightly inefficient, but good enough for this game). Next we call the draw() method of the sprite container. Since this sprite container is really an instance of the “[DrawPlain](#)” sprite group, it knows how to draw our sprites.

Lastly, we `flip()` the contents of pygame's software double buffer to the screen. This makes everything we've drawn visible all at once.

5.14 Game Over

User has quit, time to clean up.

Cleaning up the running game in *pygame* is extremely simple. In fact since all variables are automatically destroyed, we really don't have to do anything.

5.15 Assignment:

- Using the `scale` command to make the monkey smaller. Getting too good at spanking the monkey? We want to give that monkey a chance. So after every five times the monkey is hit, we want to make the monkey a bit smaller.

5.15.1 Next

Part Five

PYTHON GAME PROGRAMMING PART 5 - PARTS OF A GAME

6.1 Forgetting the details: Wrapping away implementation details.

Whenever I approach a library I am usually less interested in the library than what I can do with it. Because of this, I don't really care how the library works and what the details are. At a later date, it may become important to do some special stuff, but initially I just need it to work and most of the time this is sufficient.

When approaching game libraries, such as [PyGame](#), you essentially only care about 4 things:

6.2 Interfacing with the OS

Interfacing with the OS usually consists of creating a window, setting its properties (is it fullscreen, resizable, have a special icon, have a name) and sometimes things like getting the system time or disk access for cross platform libraries. Because Python is already cross platform, we will only need Pygame to handle creating our Window and setting it up.

Let's look at [PygameWindow](#) for how to do this.

6.3 Loading & Drawing graphics

Loading and drawing graphics is the main purpose of any game library. Some game libraries do not really do much else than this, since they expect you can fall back on simple system libraries to handle other aspects. Pygame is built on SDL which handles all aspects of interfacing with your OS and handling input, sound and graphics.

When designing your graphics wrapper, you will want to think about what features you need to be aware of. For a 2D game, as we will be designing, you should know how you will want to organize your background drawing (square tiled, isometric), how you want to layer your characters (can they overlap? how?), and anything special such as a side-scrolling game will have various layers of graphics.

For a simple but flexible wrapper, let's look at [PygameGraphics](#) to do this.

Drawing text with fonts is definitely considered in the realm of graphics, but I believe it requires enough specialized code to handle different conditions that it is worth creating a separate wrapper. Check out [PygameFont](#) for the wrapper.

6.4 Loading & Playing sound

Sound programming is usually a bit easier, as while doing complex things can be very involved, you can have a good sounding game with some very simple code.

6.5 Reading from Input devices

You can't have interactivity without reading input, and games are defined by their interactivity. I believe processing input is a layered task, and at the bottom layer you will need to poll the system devices to find out what your player is telling you.

`PygameInput` will handle thing functionality, and will let you see an overview of what your player is currently communicating to your game.

6.5.1 Next

Part Six

PYTHON GAME PROGRAMMING PART 6 - ABSTRACTION

7.1 Wrapping up the libraries and forgetting the details

Your problem: you want to play a sound, but don't know where to start with this library you are trying out.

Often libraries are much more complex than what you want to deal with.

The solution is to wrap it up in something that you are more comfortable using.

Another good reason to wrap up a library is if you want to maybe use two different libraries. For playing sounds, maybe later you want to use `openal`, or `fmod` sound libraries instead of `pygame` sound system. However you have already got lots of sound code in your programs. It is probably going to be much easier to make an implementation of your wrapper than to use try and replace all of your `pygame` specific sound code.

Why are libraries complex? Often libraries do lots of different wonderful things, for lots of different people. The libraries have lots of features, and are often quite flexible. The price for lots of features, and flexibility is often complexity.

Below we are going to make a wrapper for the `pygame` sound library as an example of wrapping up a library and forgetting the details. The `pygame` sound system is pretty simple to use already, but we're going to try making something simpler still!

7.1.1 Goals for our sounds module

One goal for making this sound wrapper will be to make it as simple as possible to load and play sounds. One line to play a sound would be acceptable (without extra lines to load the sounds).

A secondary goal for our sound module can be stated like this "Make the simple things easy, and the difficult things possible". What this means is that by default the most common things should be *really* easy to do. However we also want to make difficult things possible to do. In our case changing volume for the sounds should be possible. So should queueing sounds. Also loading files from a different directory should be possible.

7.1.2 Disadvantages to wrapping up a library

Your simplifications may make it hard to fully take advantage of certain features. This can be an easy mistake to make if you do not really know the library you are dealing with.

Wrapping is extra work you have to do. It is extra code you have to write, test, and maintain. So, are the benefits of wrapping a library enough to make it worth doing? This is up to you to decide.

7.2 Our sounds module

Here is how we will use our new, simple to use sounds module.

```
import sounds
sound_manager = sounds.SoundManager()

sound_manager.play("bump")
```

Ok so we kind of failed at our goal of making it be done in one line. However counting import lines is not fair :)

We have to have a “sounds” directory with .ogg files(compressed sound files) in it. The sound manager looks in this directory for you, and loads up all of the files.

To play the “bump” sound there would need to be a sounds/bump.ogg file.

When you call the play() method the `SoundManager` looks to see if it has already loaded the sound. If it is not loaded, it loads the sound for you.

7.2.1 Extra features for our sound module

There are a couple of extra features that are in this sound module. Which maybe you don’t need. They are hidden by using default arguments.

One feature is to allow you to say the volume of each speaker that the sound should play at.

```
# play it loud out of left, and silent in right.
sound_manager.play("bump", volume = [1., 0.] )
```

```
# play it at half volume out of both speakers.
sound_manager.play("bump", volume = [0.5, 0.5])
```

The other feature is basic sound queueing.

I’ll describe the sound queueing feature after we go through the implementation of our sounds module.

7.3 Implementing our sound module

As you may know by now a python module can simply be the code inside a .py file.

Below is our code for the sounds module in full. It should be put inside a file called sounds.py

```
import pygame
import os
import glob
import time

from pygame.locals import *

# path where all our sounds are.
SOUND_PATH = os.path.join("sounds")

def get_sound_list(path = SOUND_PATH):
    """ gets a list of sound names without thier path, or extension.
    """
    # load a list of sounds without path at the beginning and .ogg at the end.
    sound_list = map(lambda x:x[len(SOUND_PATH):-4],
                     glob.glob(os.path.join(SOUND_PATH, "*.ogg")))
    )
```

(continues on next page)

(continued from previous page)

```

    return sound_list

SOUND_LIST = get_sound_list()

class SoundManager:
    """ Controls loading, mixing, and playing the sounds.
        Having separate classes allows different groups of sounds to be
        loaded, and unloaded from memory easily.

        Usage:
            sm = SoundManager()
            sm.play("bump")
    """

    def __init__(self):
        """
        """
        # keyed by the sound name, value is a sound object.
        self.sounds = {}

        # keyed by sound name, value is the channel.
        self.chans = {}

        self._debug_level = 0

        # sounds which are queued to play.
        self.queued_sounds = []

    def _debug(self, x, debug_level = 0):
        """ Used for optionally printing debug messages.
        """
        if self._debug_level > debug_level:
            print(x)

    def load(self, names = SOUND_LIST, path = SOUND_PATH):
        """Loads sounds."""
        sounds = self.sounds

        if not pygame.mixer:
            for name in names:
                sounds[name] = None
            return
        for name in names:
            if not sounds.has_key(name):
                fullname = os.path.join(path, name+'.ogg')
                try:
                    sound = pygame.mixer.Sound(fullname)
                except:
                    sound = None
                    self._debug("Error loading sound", fullname)
                sounds[name] = sound

    def _getSound(self, name):

```

(continues on next page)

(continued from previous page)

```

    """ Returns a Sound object for the given name.
    """
    if not self.sounds.has_key(name):
        self.load([name])

    return self.sounds[name]

def play(self, name, volume=[1.0, 1.0], wait = 0):
    """ Plays the sound with the given name.
        name - of the sound.
        volume - left and right.  Ranges 0.0 - 1.0
        wait - used to control what happens if sound is allready playing:
            0 - will not wait if sound playing.  play anyway.
            1 - if there is a sound of this type playing wait for it.
            2 - if there is a sound of this type playing do not play again.
    """

    vol_l, vol_r = volume

    sound = self._getSound(name)

    if sound:
        # check to see if we want to do any sound queueing, and handle it.
        if wait in [1,2]:
            # check if the sound is allready playing, and is busy...
            if self.chans.has_key(name) and self.chans[name].get_busy():
                if wait == 1:
                    # sound is allready playing we wait for it to finish.
                    self.queued_sounds.append((name, volume, wait))
                    return
                elif wait == 2:
                    # not going to play sound if playing.  We do nothing.
                    return

            # play the sound, and store its channel in a
            # dictionary, keyed by the sound name.
            self.chans[name] = sound.play()

            # if the sound did not play, start fading out a channel, and
            # use pygame's queueing to queue up a sound on that channel.
            if not self.chans[name]:
                # forces a channel to return. we fade that out,
                # and enqueue our one.
                self.chans[name] = pygame.mixer.find_channel(1)
                self.chans[name].fadeout(100)
                self.chans[name].queue(sound)

            # if we have a channel, set its volume.
            if self.chans[name]:
                self.chans[name].set_volume(vol_l, vol_r)

def update(self, elapsed_time):
    """ This should be called frequently.  At least once every game tic/frame.
    """
    # if the sound for the channel is not busy we
    for name in self.chans.keys():
        if not self.chans[name].get_busy():

```

(continues on next page)

(continued from previous page)

```
        del self.chans[name]
        # copy the current queue, to the old queue.
        old_queued = self.queued_sounds

        # start a new queue.
        self.queued_sounds = []

        # Try and play any sounds from the old queue.
        # This may queue the sounds again, if they still shouldn't be played.
        for snd_info in old_queued:
            name, volume, wait = snd_info
            self.play(name, volume, wait)
```

Here is a little test program for our new sounds module.

```
import pygame, time
import sounds

pygame.init()
sound_manager = sounds.SoundManager()
sound_manager.play("bump")

# we sleep for one second so that pygame has time to play the sound before quitting.
time.sleep(1)
```

You can get a [bump.ogg](#) sound to play with it. Of course it's fun making your own sounds in the microphone :)

7.3.1 Next

Part Seven

PYTHON GAME PROGRAMMING PART 7 - MAKE A GAME

This lecture we are going to run through making a game. Or what some people would call a prototype of a game. It will be kind of like that old table top game starving starving rhinoceros(or something like that). Where lots of balls roll around, and you have to get your rhinoceros to eat them up.

The person who eats the most balls wins.

It is a simple game, but one with a smaller enough scope that we can finish it fairly quickly. Which are my favourite types of games to finish.

It could also be expanded later with more features if we want to.

8.1 A slightly detailed game design.

First thing we do is think about what features will be in the first version of our game. To finish the game quickly we want to make the design as simple as possible.

Why do we want to finish the game quickly? Evaluating the game quicker. If the game play isn't fun, then we want to know about it as soon as possible to fix it. Or we may even decide that the game is not possible to fix, and stop making it all together, and try out some other ideas.

So we want to limit the features to the minimal ones which are required to make the game work.

8.1.1 Limited game play features

There will be two rhinos which eat up the balls. We may want to add more rhinos later, but two will be the minimum number.

The rhinos will be facing in one direction. Maybe in a later version we will allow the rhinos to move. But for now each of them will be in the middle of the game play area, and facing opposite from each other.

Two player game to start with. As coding a fun AI can be a lot of work, we will save that for another version. Two player games are much easier to do than single player games most of the time. By two player we mean two players playing on the same computer, not network play. So one player will use the mouse, and another the keyboard. Or maybe both will use the keyboard.

Balls will bounce off the walls. So we will need to figure out how to see if a ball hits the wall, and how to make it bounce back.

Balls will also need to move around. To start with we will not take into account acceleration. All balls will move at the same speed.

Show the score for each of the rhinos. In the top left corner, and top right corner the score will be shown. The top left will be for the rhino on the left, the top right will be for the rhino on the right.

Opening and closing the rhinos mouth. When the balls come towards the players rhino, and the mouth is closed the balls will bounce off. If the mouth is open the balls will go inside the mouth. But the rhino will have to close the mouth to swallow the balls.

8.1.2 Game play uncertainty.

At this stage we have a small design of our game. But a few things are uncertain. The main one is, will this be fun?

The part which I am uncertain about is how the mouth closing, and opening to eat the balls is going to be fun. Maybe we will need to make the balls hit the back of its mouth, and bounce back? How hard should it be to eat the balls?

Another uncertainty I have is about how the balls will bounce around. I can possibly imagine that the balls could bounce around in such a way that they never go near the players rhinos mouths. Perhaps we will have to add a game play element that makes the balls go towards the players mouth.

These are just two uncertainties I have. You may have different ones, or see how these problems could easily be solved by changing the design. However both of these uncertainties will become clear as we start to develop the game, and when we finish what we have set out to do. Once the game is done, we may find that neither are problems, and that the game is fun! Yah! However we may find that they need fixing. Another more likely possibility is that there will be other problems.

So we make a minimal design, begin implementing that, and then improve the design as we go.

8.1.3 Graphical design

When we consider graphical design for a game, it is often best to start with place holder graphics. This allows us to quickly test out the game play ideas.

It is also a good idea to draw basically what your game will look like. Stick figures, and boxes are usually enough.

Here are some other factors you may want to take into account when making your graphics.

- The graphical routines you have at your disposal. Do we have a 2d engine, a 3d engine?
- The skills you have. 2d graphics, drawing, 3d modeling, animation.
- What the game play requires.
- Time you have. 2d graphics are generally quicker to do than 3d graphics.
- Your audience. What type of computer do they have?

For our game I don't want to do really advanced graphics to start with. I want to get it done quickly, so I'll choose to use 2d graphics. I also don't want to use any external images to start with, so I will use the `pygame.draw` graphics routines. These allow you to draw lines, circles, rectangles, and flat polygons.

Not using any external files will mean that I will be able to make the game work with one python script.

Once the game is complete, if I wish to continue with it, I could improve the graphics.

Drawing the game design

As you can see I spent maybe two minutes on this drawing.

You can draw it on paper, or on the computer. Just a rough draft is necessary for the game design. You may want to spend more time on it drawing up different rhinos at different sizes, and different frames of their animation.

As we are not going to do much graphics to start with, just concentrate on getting the shapes and sizes of things correct.

8.2 Sound design

No sounds to start with. Later I may want all manner of sounds. But as the game design so far does not concerntrate on sounds as game play, I will not use any sounds.

If I were to do sounds, some could be:

- ball rolling. Get a marble and roll it along a desk.
- ball hitting the wall. Roll a marble into something.
- Rhino noises. Hrmmm, these would be harder to make. Be creative!

8.3 User input design

Some people say the user interface to your game is the most important part. There are a few reasons they say this. One is that if the player can't figure out the controls, they will not play. Also it is what the player will be doing.

For our game it will only require one input. That is all the player can do is to say open mouth, or close mouth. That's it! Simple eh? Maybe too simple. But it will do for now.

There are lots of things we could add later, but opening, and closing the mouth of the rhino is the main interaction the player will have. So we want this main interaction to also be the easiest to find out how to do.

Clicking the mouse button, and pressing any key will be the way to control our game. Player one will control the first rhino(one on left) with the mouse click. Player two will control its rhino with pressing a key on the keyboard.

The user input to the game may also change as we make the game. So as you make a game, occasionally evaluate how your controls will work. Especially after adding lots more user interaction.

8.3.1 Other input devices

We will not put into our game support for any more types of input. But many of them could be added later quite easily. However then we would need to detect or allow the player to chose which input they are using.

We could make our game see if the player is pressing on a joystick and make the joystick controll player two. Or perhaps make it control player one. In that way we could avoid making the player manually choose which input they want to use. It will just use which ever input they bash on :)

8.3.2 timing and user input.

We may want to limit how fast a person can do things. In our game we will probably want to limit how quickly the player can open and close the mouth of the rhino.

The game could be quite different if we make it so that you can only open and close the mouth every two seconds. Compared to if you could open and close the mouth as quickly as you could click.

8.3.3 Other things player can do.

Two other things most games have are:

- exiting/quitting the game.
- pausing the game.

8.3.4 User input for quitting

There are some conventions in games for quitting. One is the 'q' key. Another is the ESC key. Often the esc or q keys take the player to the main menu. As we will not have a main menu those keys will just quit the game. Another convention is to use ALT+F4. But that is not

Quitting with the mouse is another thing we may want to add. Maybe a big quit button somewhere on the screen. In the first version we won't implement it, as it is too much work to test out the game play. Later though it may

be important. One problem with that is that the player may accidentally press it. If we make our game go in windowed mode, the player can quit with the mouse by using the quit button on the window.

8.3.5 User input for pausing.

It is generally a good idea to build pausing into the game as early as possible. As it can be a pain to add in later.

The convention for pausing a game is usually the 'p' key. Also some games pause the game when you press ESC, whilst they show a menu.

8.4 Code design

First we need to decide what our audience is. From that we decide which programming language, methods, and libraries to use.

In this case the audience is those people reading a python for programming set of lectures/articles. So our language, and library choice is python, and pygame.

As we want the game to be easy to run for you, I've decided to make the game in one file. So all our graphics will be done with code, and there will be no sound files with the game.

Because we want to finish the game quickly we choose to do a 2d game.

Our game design will probably not require any super speed code.

The only performance problems I could see us having is with lots of balls moving around the screen. If we use the `pygame.sprite` classes this shouldn't be a problem. Also the sprite classes will help us organise the different visual elements of our game.

Different visual elements in the game:

- Score. In the corners of the screen. Will change the number when the rhino scores.
- Rhinos. Two of them in the middle of the screen. These will be animated.
- Balls. There will be many of them. They are allways moving, except when eaten.

A slightly hard bit of code identified earlier in game play uncertainty, is the ball bouncing code. Luckily there have been lots of games done about balls bouncing around so the techniques to do so are described in lots of websites. If we have a problem coding it, with a little research we should find the answer.

If the game starts to run a little slowly when we have lots of balls, we could just reduce the number of balls moving around.

Also for a bit of variation, we may want to give the players the option of changing the amount of balls in play. Although we won't give them the option in the first version of the game.

8.5 Coding the game

Now we dive into the coding of the game. I like to do it in small steps. First get the screen up. Then draw the basic elements to the screen. Then maybe add in some keyboard handling, and some mouse handling. Below I describe the process I take making the game.

8.5.1 Getting something on screen

I'll make some basic code to get a few things on screen first.

So I initialize things, make a main loop, and put in the event handling for quitting.

```

import pygame
from pygame.locals import *

pygame.init()

display_flags = DOUBLEBUF
width, height = 640, 480

if pygame.display.mode_ok((width, height), display_flags):
    screen = pygame.display.set_mode((width, height), display_flags)

run = 1

clock = pygame.time.Clock()

while run:

    events = pygame.event.get()

    for event in events:

        if event.type == QUIT or (event.type == KEYDOWN and
                                   event.key in [K_ESCAPE, K_q]):
            # set run to 0 makes the game quit.
            run = 0

        # add the game play in here later.

    pygame.display.flip()

    # limit the game to about 40fps, or 40 ticks per second.
    clock.tick(40)

```

Ok. So now I have a black screen showing in a window, which I can quit from. I limited the frame rate to 40 frames per second so when I do animation, it is smoother. For smooth animation you need as constant a frame rate as possible. Which is one of the reasons why tv, and film run at 24 or 30 frames per second. There is more to it than this. If you want to know more there is a good discussion on the topic at <http://ludumdare.com/forums/viewtopic.php?topic=141&forum=2&22>

8.5.2 Drawing the balls

For the balls I am simply going to use the `pygame.draw.circle` function. To draw a filled in circle.

We need two different colored circles, so this little function will draw them to two surfaces for us.

```

import pygame
from pygame.locals import *

def render_ball_simple(radius, color):
    """ Returns (surf, rect) containing a picture of
        a circle of the radius, and color given.
    """
    size = radius * 2
    surf = pygame.Surface((size, size))
    pygame.draw.circle(surf, color, (radius, radius), radius)
    return surf, surf.get_rect()

def max(x, y):

```

(continues on next page)

(continued from previous page)

```

""" returns x, unless x > y.  if it is it returns y.
    """
    if x > y:
        return y
    else:
        return x

def render_ball_funky(radius, color):
    """ Returns (surf,rect) containing a picture of
        a slightly shaded ball of the radius, and color given.
        """

    size = radius * 2
    surf = pygame.Surface((size, size))

    # we progressively draw smaller circles of different colors.
    increment = int(radius / 4)
    for x in range(4):
        iradius = radius - (x * increment)
        print(iradius)
        isize = iradius * 2
        icolor = [0, 0, 0]

        # we increment the color.  if it is bigger than 255 we make it 255.
        icolor[0] = max(color[0] + (x * 15), 255)
        icolor[1] = max(color[1] + (x * 15), 255)
        icolor[2] = max(color[2] + (x * 15), 255)

        pygame.draw.circle(surf, icolor, (radius, radius), iradius)

    return surf, surf.get_rect()

def render_ball(radius, color):
    """ Returns (surf,rect) containing a picture of
        a ball of the radius, and color given.
        """
    # we use the kind of funk one...
    return render_ball_funky(radius, color)

def main():

    pygame.init()

    display_flags = DOUBLEBUF
    width, height = 640, 480

    if pygame.display.mode_ok((width, height), display_flags):
        screen = pygame.display.set_mode((width, height), display_flags)

    run = 1

    clock = pygame.time.Clock()

    # draw some graphics to surfaces.

```

(continues on next page)

(continued from previous page)

```

ball1,ball1_rect = render_ball_funky(10, (50, 200, 200))
ball2,ball2_rect = render_ball_simple(6, (50, 200, 200))

# move the simple one towards the center top of the screen.
ball2_rect.x = 200

while run:

    events = pygame.event.get()

    for event in events:

        if event.type == QUIT or (event.type == KEYDOWN and
                                   event.key in [K_ESCAPE, K_q]):
            # set run to 0 makes the game quit.
            run = 0

        # add the game play in here later.
        screen.blit(ball1, ball1_rect)
        screen.blit(ball2, ball2_rect)

    pygame.display.flip()

    # limit the game to about 40fps, or 40 ticks per second.
    clock.tick(40)

# this runs the main function if this script is called to run.
# If it is imported as a module, we don't run the main function.
if __name__ == "__main__":
    main()

```

In this code you can see that I have drawn two ball looking things to the screen. Circles really. I got carried away with the draw ball function, and made a `render_ball_funky()`. It draws four circles progressively lighter to give a really simple shading effect. This is what we call feature creep. I just did it for fun.

Feature creep gets in the way of you finishing the game. So avoid it! ;)

I have also moved the mainloop and the initialisation code inside a main function. This is just to make it a bit neater.

Inside the main loop(the while loop), I do two blits. One for each of the balls. The one in the center top of the screen is the simple circle, the one on the left top is the so called funky ball. If you recall a blit means to draw, or to copy an image. In this case we are blitting directly to the screen.

The max function we define makes sure that the color values don't get above 255. It is what some people call a helper function. That is a small function made to make your code a bit nicer looking.

You should try and run this code as it gets developed. Add in some print functions, and play around with it, so you can get the flow of how it is working.

PIXEL PERFECT COLLISION DETECTION IN PYGAME WITH MASKS.

“BULLSHIT! That bullet didn’t even hit me!” they cried as the space ship starts to play the destruction animation, and Player 1 life counter drops by one. Similar cries of BULLSHIT! are heard all over the world as thousands of people lose an imaginary life to imperfect collision detection every day.

Do you want random people on the internet to cry bullshit at your game? Well do ya punk?

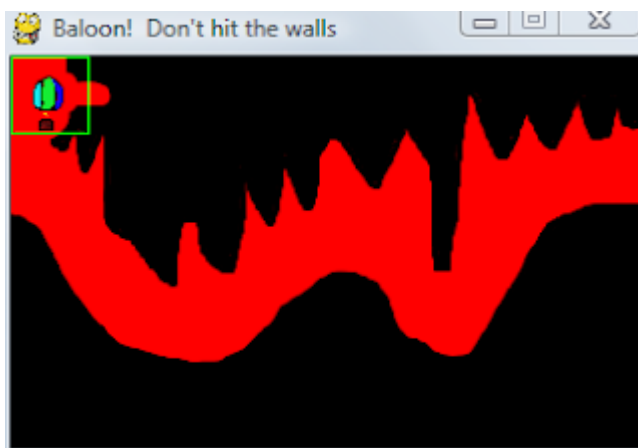
Bounding boxes are used by many games to detect if two things collide. Either a rectangle, a circle, a box or a sphere are used as a crude way to check if two things collide. However for many games that just isn’t enough. Players can see that something didn’t collide, so they are going to be crying foul if you just use bounding boxes.

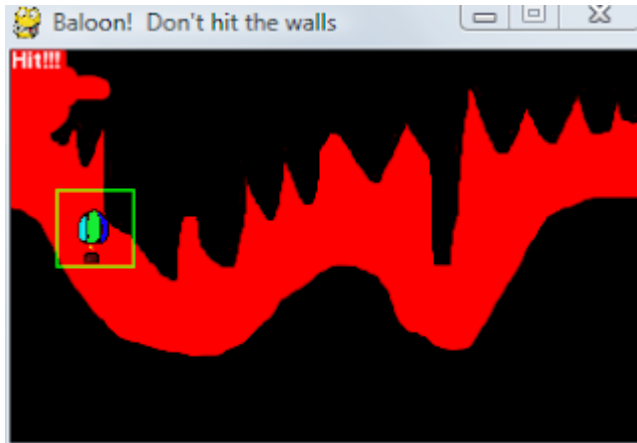
Pygame added fast and easy pixel perfect collision detection. So no more bullshit collisions ok?

Code to go along with this article can be found here (https://github.com/illume/pixel_perfect_collision).

9.1 Why rectangles aren’t good enough.

Here are some screen shots of a little balloon game I made modeled after an old commodore 64 game I typed in when I was eight. Here you can see a balloon, and a cave. The idea is you have to move the baloon through the cave without hitting the walls. Now if you used just bounding rectangle collisions, you will see how it would not work, and how the game would be no fun - because the rectangle(drawn in green around the balloon) would hit the sides when the balloon didn’t really hit the sides.





You can download the balloon mini game code to have a look at with this article.

9.2 How is pixel perfect collision detection done? Masks.

Instead of using 8-32bits per pixel, pygame's masks use only 1 bit per pixel. This makes it very quick to check for collisions. As you can compare 32 pixels with one integer compare. Masks use bounding box collision first - to speed things up.

Even though bounding boxes are a crude approximation for collisions, they are faster than using bitmasks. So pygame first does a check to see if the rectangles collide - then if the rectangles do collide, only then does it check to see if the pixels collide.

9.3 How to use pixel perfect collision detection in pygame?

There are a couple of ways you can use pixel perfect collision detection with pygame.

- Creating masks from surfaces.
- Using the `pygame.sprite` classes.

You can create a mask from any surfaces with transparency. So you load up your images normally, and then create the masks for them.

Or you can use the `pygame.sprite` classes, which handle some of the complexity for you.

9.4 `Mask.from_surface` with Alpha transparency.

By default pygame uses either color keys, or per pixel alpha values to see which parts of an image are converted into the mask.

Color keyed images have either 100% transparent or fully visible pixels. Whereas per pixel alpha images have 255 levels of transparency. By default pygame uses 50% transparent pixels as on, or ones that are to collide with.

It's a good idea to pre-calculate the mask, so you do not need to generate it every frame.

9.5 Checking if one mask overlaps another mask.

It is fairly simple to see if one mask overlaps another mask.

Say we have two masks (a and b), and also a rect for where each of the masks is.

```
#We calculate the offset of the second mask relative to the first mask.
offset_x = a_rect[0] - b_rect[0]
offset_y = a_rect[1] - b_rect[1]
# See if the two masks at the offset are overlapping.
overlap = a.overlap(b, (offset_x, offset_y))
if overlap:
    print "the two masks overlap!"
```

9.6 Pixel perfect collision detection with pygame.sprite classes.

The pygame.sprite classes are a high level way to display your images. They provide things like collision detection, layers, groups and lots of other goodies.

Note: balloon2.py that comes with this article uses sprites with masks.

If you give your sprites a .mask attribute then they can use the built in collision detection functions that come with pygame.sprite.

```
class Balloon(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self) #call Sprite initializer
        self.image, self.rect = pygame.image.load("balloon.png")
        self.mask = pygame.mask.from_surface(self.image)

b1 = Balloon()
b2 = Balloon()

if pygame.sprite.spritecollide(b1, b2, False, pygame.sprite.collide_mask):
    print "sprites have collided!"
```

9.7 Collision response - approximate collision normal.

Once two things collide, what happens next? Maybe one of these things...

One of the things blows up, disappears, or does a dying animation.

Both things disappear.

Both things bounce off each other.

One thing bounces, the other thing stays. If something going to be bouncing, and not just disappearing, then we need to figure out the direction the two masks collided. This direction of collision we will call a collision normal.

Using just the masks, we can not find the exact collision normal, so we find an approximation. Often times in games, we don't need to find an exact solution, just something that looks kind of right.

Using an offset in the x direction, and the y direction, we find the difference in overlapped areas between the two masks. This gives us the vector (dx, dy), which we use as the collision normal.

If you understand vector maths, you can add this normal to the velocity of the first moving object, and subtract it from the other moving object.

```
def collision_normal(left_mask, right_mask, left_pos, right_pos):

    def vadd(x, y):
        return [x[0]+y[0],x[1]+y[1]]

    def vsub(x, y):
        return [x[0]-y[0],x[1]-y[1]]

    def vdot(x, y):
        return x[0]*y[0]+x[1]*y[1]

    offset = list(map(int, vsub(left_pos, right_pos)))

    overlap = left_mask.overlap_area(right_mask, offset)

    if overlap == 0:
        return

    """Calculate collision normal"""

    nx = (left_mask.overlap_area(right_mask, (offset[0]+1,offset[1])) -
          left_mask.overlap_area(right_mask, (offset[0]-1,offset[1])))
    ny = (left_mask.overlap_area(right_mask, (offset[0],offset[1]+1)) -
          left_mask.overlap_area(right_mask, (offset[0],offset[1]-1)))
    if nx == 0 and ny == 0:
        """One sprite is inside another"""
        return

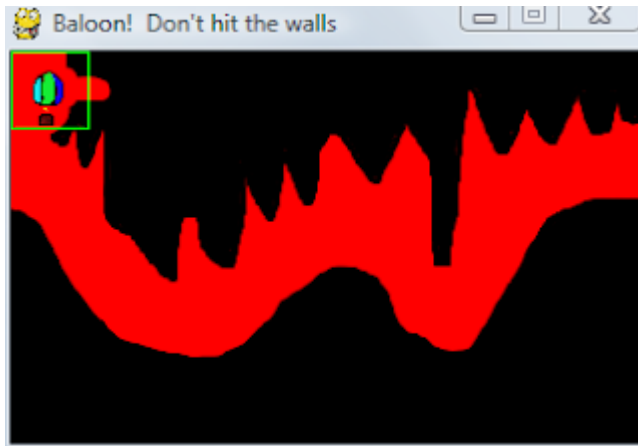
    n = [nx, ny]

    return n
```

9.8 Fun uses for masks.

Here's a few fun ideas that you could implement with masks, and pixel perfect collision detection.

- A balloon game, where the bit masks are created from nicely drawn levels - which are then turned into bitmasks for pixel perfect collision detection. No need to worry about slicing the level up, or manually specifying the collision rectangles, just draw the level and create a mask out of it. Here's a screen shot from the balloon code that comes with this article:



- A platform game where the ground is not made out of platforms, so much as pixels. So you could have curvy ground, or single pixel things the characters could stand on.
- Mouse cursor hit detection. Turn your mouse cursor into something, and rather than have a single pixel hit, instead have the hit be any pixel under the mouse cursor.
- “Worms” style exploding terrain.

YOU KNOW WHAT'S AWESOME? YOU ARE!

(Also this stuff)

10.1 awesome libraries for pygame

- available via pip
- documented (with examples, and API docs)
- generally awesome for some reason (even if not perfect, maybe it has potential)

Where to begin? ...

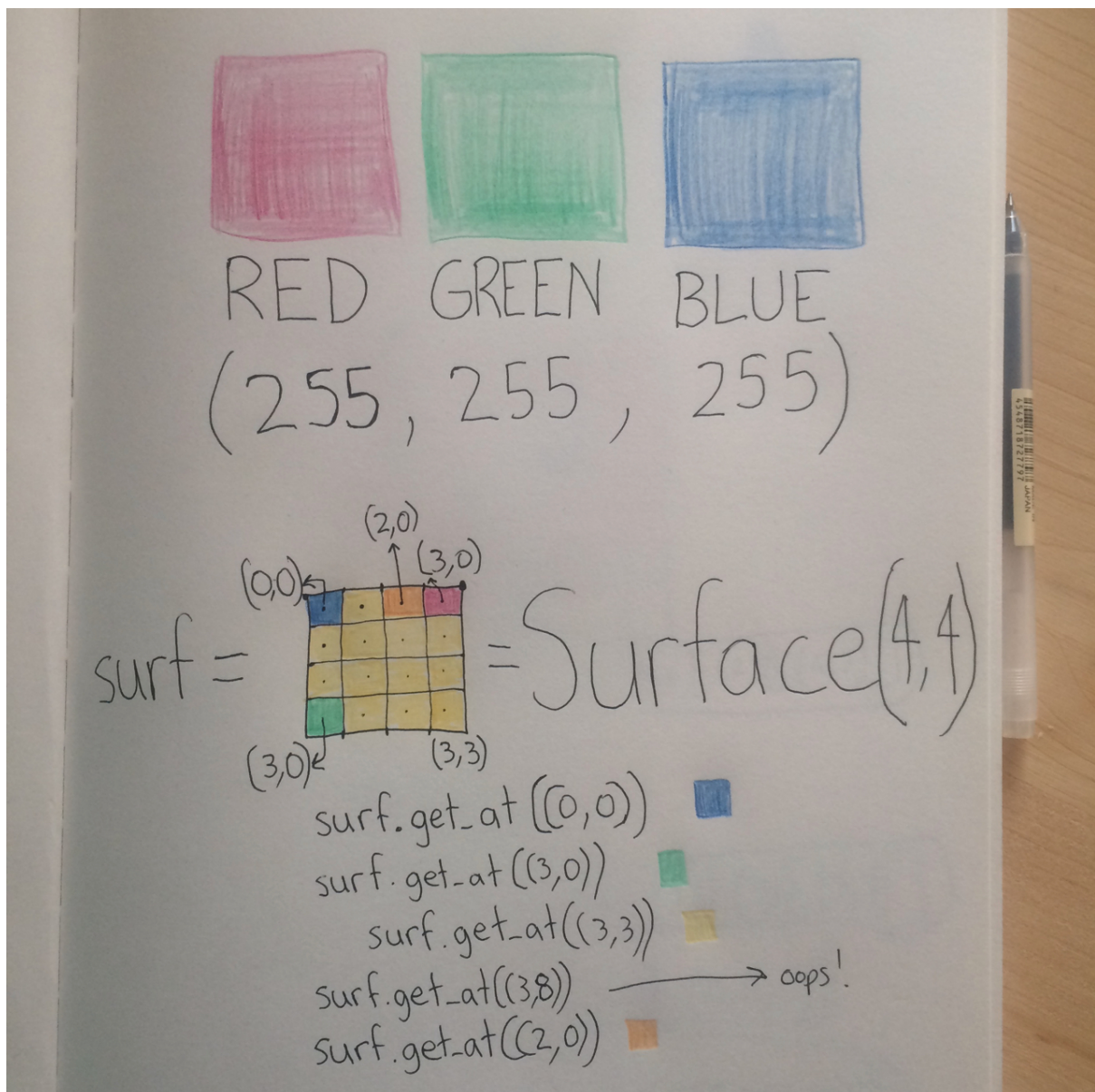
- [mu-editor](#) - an editor for beginners. It also comes bundled with pygame and pgzero. The main idea of Mu is that you use a simplified but real editor to begin with, and then to leave it behind once you advance.
- [Thonny](#) - is another python editor for beginners.
- [A Little Bit of Widgetry for PyGame](#) - A GUI library for pygame. This is a fork of the original Albow library, but it works with python 3 and has been updated since then.
- [Thorpy](#) - a GUI library for pygame.
- [pymunk](#) - 2d physics library. Lots of examples, very well maintained.
- [pyscroll](#) - Scrolling maps. Fast scrolling images easy.
- [pyTMX](#) - Reads Tiled Map Editors TMX maps. Can use a Map editor easily.
- [pyinstaller](#) - Make installers for different platforms. Very easy to use to make installers.
- [animation](#) - Tasks and tweening using pygame groups. No framework needed to smoothly move sprites or execute things over time.
- [symplehfs](#) - hierarchial state machine.
- [spritesheetlib](#) - loading sprite sheets.
- [pyknic](#) - collection of useful tools (tweening, animation, context, timing, fonts, spritesystem, skeleton, ...) for games.
- [pytmxloader](#) - Map loader for tmx files
- [pygame-text](#) - Greatly simplifies drawing text with the pygame.font module.
- [pygame-menu](#) - A menu for pygame, simple, lightweight and easy to use.
- [TextBoxify](#) - package to easily create dialog boxes for games. Want to show a character talking, with a box of text next to it? Then this is for you.
- [pgzero](#) - a framework for games built on top of pygame. Intended for education.
- [Wasabi2d](#) - another game framework by lordmauve, this time it's cutting edge.
- [moderngl](#) - Modern OpenGL bindings for python.

- [PyGLM on GitHub](#) comes with perlin and simplex noise (2D, 3D and 4D), matrix transformations, quaternions, vectors and other GLSL type goodies.

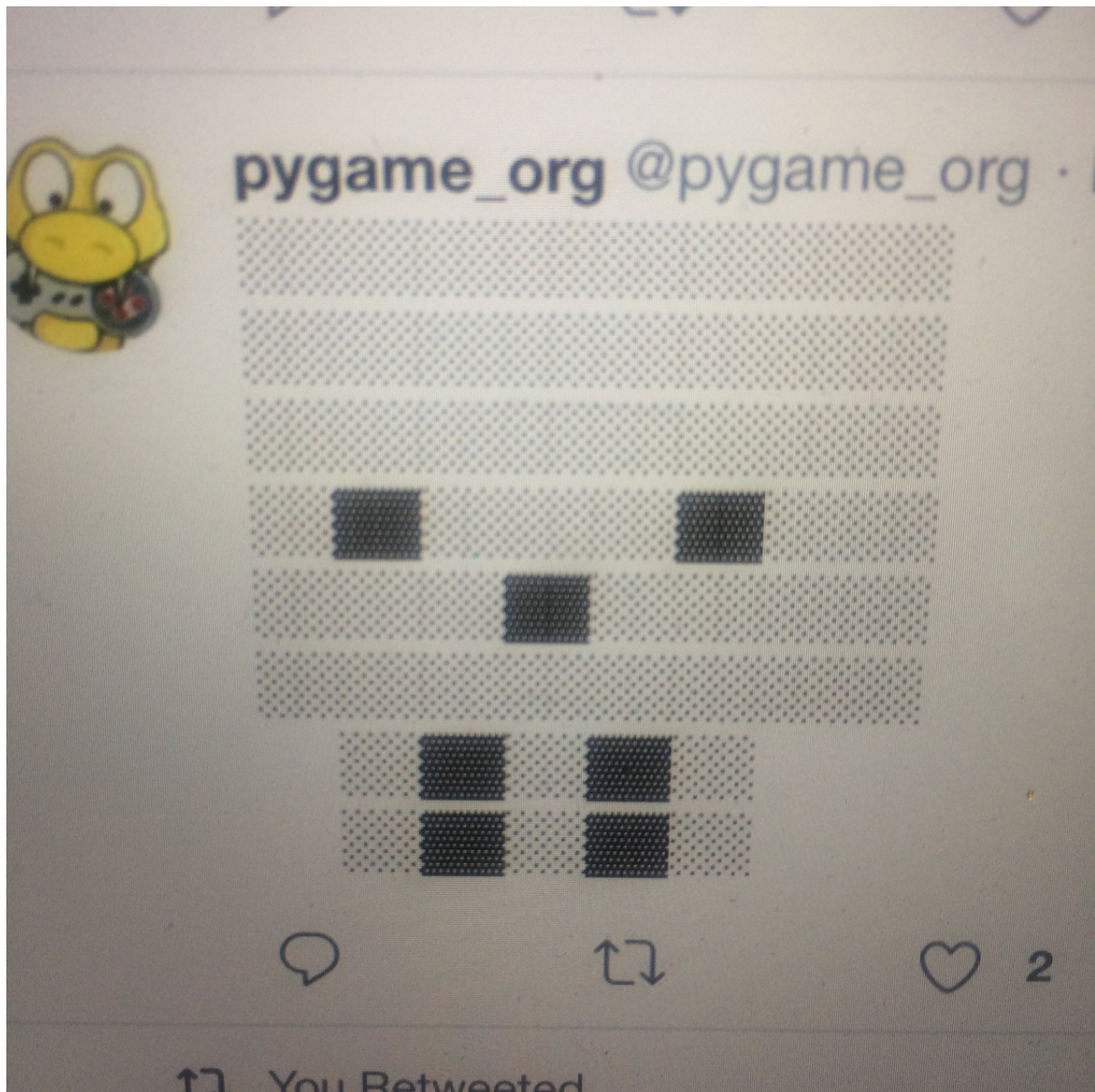
SDL2 BASICS TUTORIAL FUNDAMENTALS IN C

Until text comdes, see “./code/hey.c”

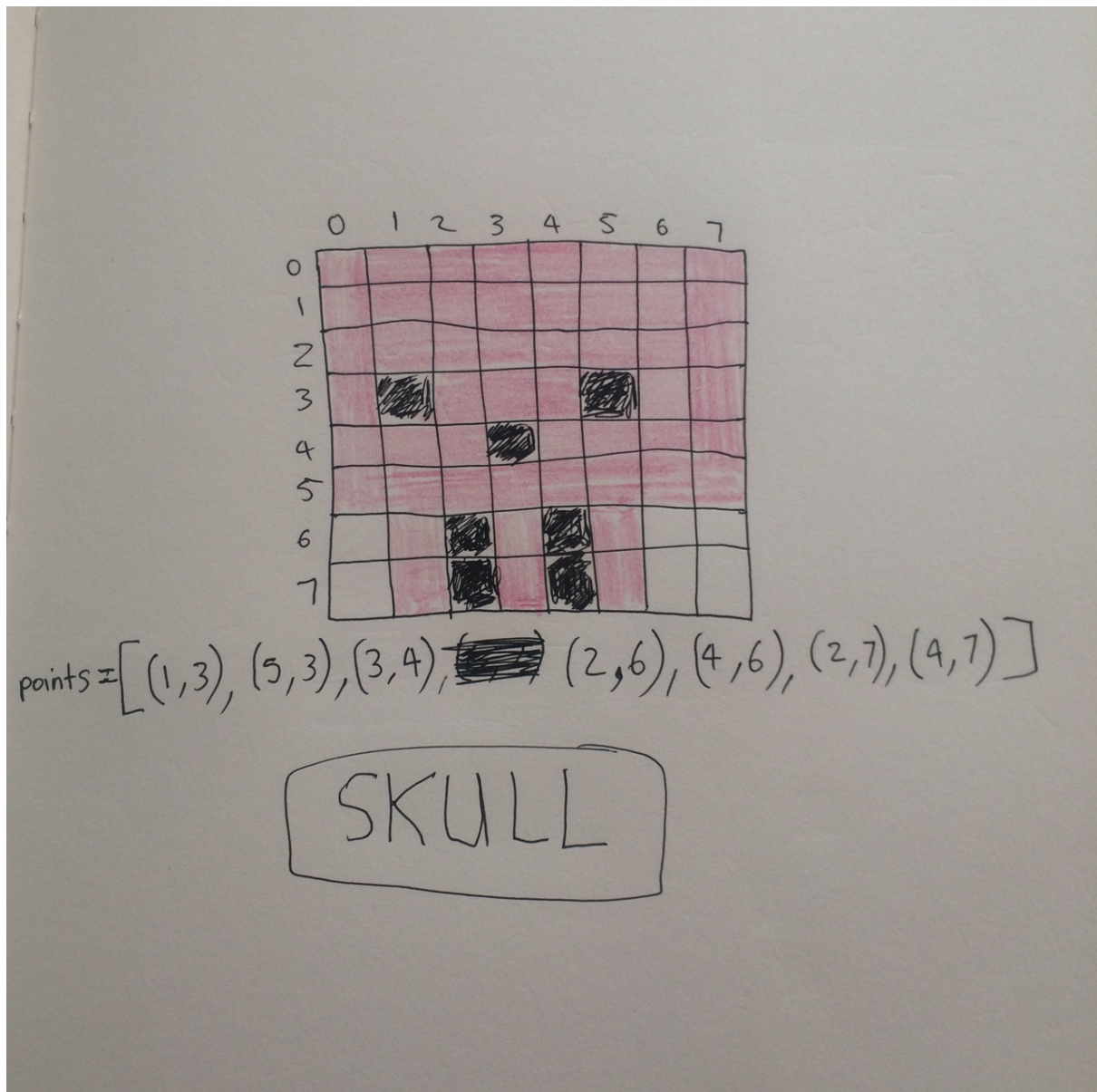
This is a sentence.

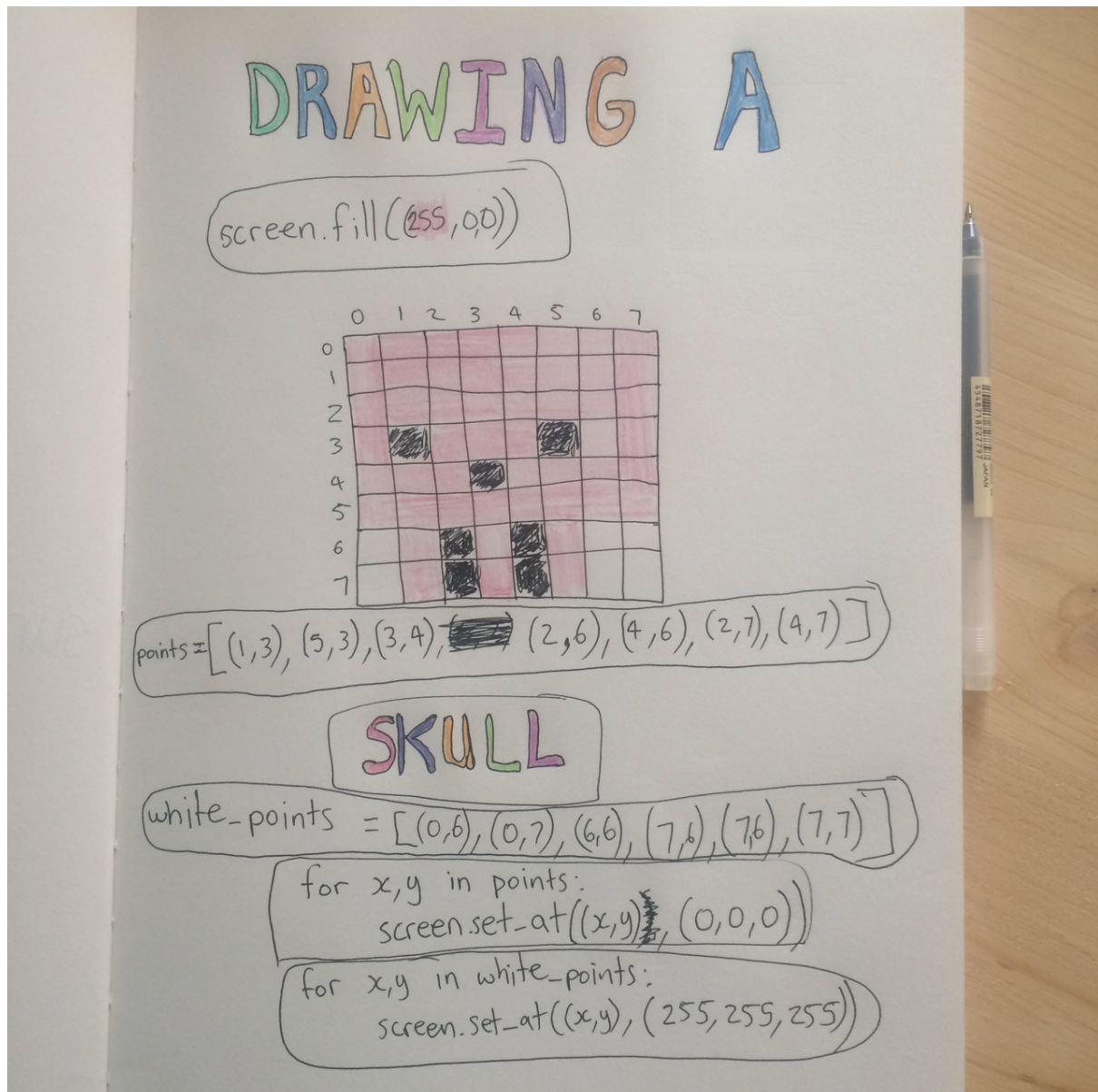


This is a sentence2.



This is a sentence³.





Some more stuff.

POST MODERN C TOOLING

Contemporary C tooling for making higher quality C, faster or more safely.

In 2001 or so people started using the phrase “Modern C++”. So now that it’s 2019, I guess we’re in the post modern era? Anyway, this isn’t a post about C++ code, but some of this information applies there too.



Fig. 1: No logo, but it’s used everywhere.

12.1 Welcome to the post modern era.

Some of the C++ people have pulled off one of the cleverest and sneakiest tricks ever. They required ‘modern’ C99 and C11 features in ‘recent’ C++ standards. Microsoft has famously still clung onto some 80s version of C with their compiler for the longest time. So it’s been a decade of hacks for people writing portable code in C. For a while I thought we’d be stuck in the 80s with C89 forever. However, now that some C99 and C11 features are more widely available in the Microsoft compiler, we can use these features in highly portable code (but forget about C17/C18 ISO/IEC 9899:2018/C2X stuff!!). Check out the “New” [Features in C](#) talk, and the [Modern C](#) book for more details.

So, we have some pretty new language features in C with C11. But what about tooling?

12.2 Tools and protection for our feet.

C, whilst a work horse being used in everything from toasters, trains, phones, web browsers, ... (everything basically) - is also an excellent tool for shooting yourself in the foot.

Noun

footgun (*plural* footguns)

1. (informal, humorous, derogatory) Any **feature** whose addition to a product results in the user shooting themselves in the foot. C.

Tools like linters, test coverage checkers, static analyzers, memory checkers, documentation generators, thread checkers, continuous integration, nice error messages, ... and such help protect our feet.

How do we do continuous delivery with a language that lets us do the most low level footgunie things ever? On a dozen CPU architectures, 32 bit, 64bit, little endian, big endian, 64 bit with 32bit pointers (wat?!?), with multiple compilers, on a dozen different OS, with dozens of different versions of your dependencies?

Surely there won't be enough time to do releases, and have time left to eat my vegan shaved ice desert after lunch?

12.3 Debuggers

Give me 15 minutes, and I'll change your mind about GDB. – <https://www.youtube.com/watch?v=PorfLSr3DDI>

Firstly, did you know gdb had a curses based 'GUI' which works in a terminal? It's a quite a bit easier to use than the command line text interface. It's called TUI. It's built in, and uses emacs key bindings.

But what if you are used to VIM key bindings? **cgdb** to the rescue.

VIM has integrated gdb debugging with **TermDebug** since version 8.1.

Also, there's a fairly easy to use web based front end for GDB called **gdbgui**

(<https://www.gdbgui.com/>). For those who don't use an IDE with debugging support built in (such as Visual studio by Microsoft or XCode by Apple).

12.3.1 Reverse debugger

Normally a program runs forwards. But what about when you are debugging and you want to run the program backwards?

Set breakpoints and data watchpoints and quickly reverse-execute to where they were hit.

How do you tame non determinism to allow a program to run the same way it did when it crashed? In C and with threads some times it's really hard to reproduce problems.

```

1505 int
1506 main (int argc, char *argv[])
1507 {
1508     /* Uncomment to debug and attach */
1509     #if 0
1510     int c;
1511     read (0, &c, 1);
1512     #endif
1513
1514     parse_long_options (targc, &argv);
1515
1516     current_line = ibuf_init ();
1517
1518     if (create_and_init_pair () == -1)
1519     {
1520         fprintf (stderr, "Es:Xd Unable to create PTY pair", _FILE_, _LINE);
1521         exit (-1);
1522     }
1523
1524     /* First create tgdb, because it has the error log */
1525     if (start_gdb (argc, argv) == -1)
1526     {
1527
/home/mike/dev/cgdb/cgdb/src/cgdb.c
0x00002b4ea5b2fa33 in select () from /lib/libc.so.6
(tgdb)
Breakpoint 2 at 0x4041c1: file cgdb.c, line 1514.
(tgdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
The program being debugged has been started already.
Starting program: /home/mike/dev/cgdb/cgdb/src/cgdb

Breakpoint 2, main (argc=0, argv=0x7ffff8065720) at cgdb.c:1516
(tgdb) p argc
$1 = 0
(tgdb)

```

Fig. 2: <https://cgdb.github.io/>

rr helps with this. It's actual magic.

<https://rr-project.org/>

12.3.2 LLDB - the LLVM debugger.

Apart from the ever improving `gdb`, there is a new debugger from the LLVM people - `lldb` (<https://lldb.llvm.org/>).

12.3.3 IDE debugging

Visual Studio by Microsoft, and XCode by Apple are the two heavy weights here.

The free Visual Studio Code also supports debugging with GDB.

<https://code.visualstudio.com/docs/languages/cpp>

Sublime is another popular editor, and there is good GDB integration for it too in the [SublimeGDB](https://packagecontrol.io/packages/SublimeGDB) package (<https://packagecontrol.io/packages/SublimeGDB>).

12.3.4 Windows debugging

Suppose you want to do post mortem debugging? With `procdump` and `WinDbg` you can. “How to take a `procdump`” (https://blogs.msdn.microsoft.com/webdav_101/2018/03/20/how-to-take-a-procdump/) is a nice tutorial on how to use `procdump`.

Launch a process and then monitor it for exceptions:

```
C:> procdump -e 1 -f "" -x c:\dumps consume.exe
```

This makes some process dumps when it crashes, which you can then open with `WinDbg` (<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-using-windbg-preview>).

12.4 Portable building, and package management

C doesn't have a package manager. . . or does it?

Ever since Debian dpkg, Redhat rpm, and Perl started doing package management in the early 90s people world wide have been able to share pieces of software more easily. Following those systems, many other systems like Ruby gems, JavaScript npm, and Pythons cheese shop came into being. Allowing many to share code easily.

But what about C? How can we define dependencies on different 'packages' or libraries and have them compile on different platforms?

How do we build with Microsofts compiler, with gcc, with clang, or Intels C compiler? How do we build on Mac, on Windows, on Ubuntu, on Arch linux? Sometimes we want to use an Integrated Development Environment (IDE) because they provide lots of nice tools. But maybe also three IDEs (XCode, Microsoft Visual C, CLion, . . .) depending on platform. But we don't want to have to keep several IDE project files up to date. But we also want to integrate nicely with different OS packagers like Debian, FreeBSD. We want people to be able to use apt-get install for their dependencies if they want. We also want to cross compile code on our beefy workstations to work on microcontrollers or slower low memory systems (like earlier RaspberryPi systems).

12.4.1 The Meson Build System.

If CMake is modern, then [The Meson Build System \(https://mesonbuild.com/index.html\)](https://mesonbuild.com/index.html) is post modern.

"Meson is an open source build system meant to be both extremely fast, and, even more importantly, as **user friendly as possible**.

The main design point of Meson is that every moment a developer spends writing or debugging build definitions is a second wasted. So is every second spent waiting for the build system to actually start compiling code."

It's first major user was GStreamer, a multi platform multimedia toolkit which is highly portable. Now it is especially popular in the FreeDesktop world with projects like gstreamer, GTK, and systemd amongst many others using it.

The documentation is excellent, and it's very fast compared to autotools or cmake.

<https://www.youtube.com/watch?v=SCZLnopmYBM> <https://www.youtube.com/watch?v=gHdTzdXkhRY>

Example meson.build for example project [polysnake\(https://github.com/jpakkane/polysnake\)](https://github.com/jpakkane/polysnake): "A Python extension module that uses C, C++, Fortran and Rust?"

```
project('polysnake', 'c', 'cpp', 'rust', 'fortran', default_options : ['cpp_std=c++14'], license
: 'GPL3+') py3_mod = import('python3') py3_dep = dependency('python3') # Rust integration
is not perfect. rustlib = static_library('func', 'func.rs') py3_mod.extension_module('polysnake',
'polysnake.c', 'func.cpp', 'ffunc.f90', link_with : rustlib, dependencies : py3_dep)
```

IDEs are supported by exporting to XCode+Visual Studio, and they provide their own interface (which a few less well known IDEs are starting to use).

12.4.2 Conan package manager

There are several packaging tools for C these days, but one of the top contenders is [Conan \(https://conan.io/\)](https://conan.io/).

"Conan, the C / C++ Package Manager for Developers The open source, decentralized and multi-platform package manager to create and share all your native binaries."

What does a CMake conan project look like? (<https://github.com/conan-io/hello>)

What does a conan Meson project look like? (https://docs.conan.io/en/latest/reference/build_helpers/meson.html)

12.4.3 CMake

“Modern CMake” is the build tool of choice for many C projects.

Just don’t read the official docs, or the official book - they’re quite out of date.

An Introduction to Modern CMake (<https://cliutils.gitlab.io/modern-cmake/>)

CGold: The Hitchhiker’s Guide to the CMake (<https://cgold.readthedocs.io/en/latest/>)

“CMake is a meta build system. It can generate real native build tool files from abstracted text configuration. Usually such code lives in `CMakeLists.txt` files.”

Around 2015-2016 a bunch of IDEs got support for CMake: [Microsoft Visual Studio](#), [CLion](#), [QtCreator](#), [KDevelop](#), and [Android Studio \(NDK\)](#). And a lot of people tried extra hard to like it, and a lot of C projects started supporting it.

Apart from wide IDE support, it is also supported quite well by package managers like VCPkg and Conan.

12.5 Interpreter and REPL

Usually C is compiled.

Bic is an interpreter for C (<https://github.com/hexagonal-sun/bic>).

Additionally there is “Cling” which is based on the LLVM infrastructure and can even do C++.

<https://github.com/root-project/cling>

12.6 Testing coverage.

Tests let us know that some certain function is running ok. Which code do we still need to test?

`gcov`, a tool you can use in conjunction with GCC to test code coverage in your programs.

`lcov`, LCOV is a graphical front-end for GCC’s coverage testing tool `gcov`.

Instructions from codecov.io on how to use it with C, and clang or gcc. (codecov.io is free for public open source repos).

<https://github.com/codecov/example-c>

Here’s documentation for how CPython gets coverage results for C.

<https://devguide.python.org/coverage/#measuring-coverage-of-c-code-with-gcov-and-lcov>

Here is the CPython Travis CI configuration they use.

<https://github.com/python/cpython/blob/master/.travis.yml#L69>

```
- os: linux
  language: c
  compiler: gcc
  env: OPTIONAL=true
  addons:
```

(continues on next page)

(continued from previous page)

```

apt:
  packages:
    - lcov
    - xvfb
before_script:
  - ./configure
  - make coverage -s -j4
  # Need a venv that can parse covered code.
  - ./python -m venv venv
  - ./venv/bin/python -m pip install -U coverage
  - ./venv/bin/python -m test.pythoninfo
script:
  # Skip tests that re-run the entire test suite.
  - xvfb-run ./venv/bin/python -m coverage run --pylib -m test --fail-env-
↪changed -uall,-cpu -x test_multiprocessing_fork -x test_multiprocessing_
↪forkserver -x test_multiprocessing_spawn -x test_concurrent_futures
  after_script: # Probably should be after_success once test suite updated to run_
↪under coverage.py.
  # Make the `coverage` command available to Codecov w/ a version of Python that_
↪can parse all source files.
  - source ./venv/bin/activate
  - make coverage-lcov
  - bash > (curl -s https://codecov.io/bash)

```

12.7 Static analysis

“Static analysis has not been helpful in finding bugs in SQLite. More bugs have been introduced into SQLite while trying to get it to compile without warnings than have been found by static analysis.” – <https://www.sqlite.org/testing.html>

According to David Wheeler in “How to Prevent the next Heartbleed”

(<https://dwheeler.com/essays/heartbleed.html#static-not-found> the security problem with a logo, a website, and a marketing team) only one static analysis tool found the Heartbleed vulnerability before it was known. This tool is called CQual++. One reason for projects not using these tools is that they have been (and some still are) hard to use. The LLVM project only started using the clang static analysis tool on its [own projects recently](#) for example. However, since Heartbleed in 2014 tools have improved in both usability and their ability to detect issues.

I think it’s generally accepted that static analysis tools are incomplete, in that each tool does not guarantee detecting every problem or even always detecting the same issues all the time. Using multiple tools can therefore be said to find multiple different types of problems.

12.7.1 Compilers are kind of smart

The most basic of static analysis tools are compilers themselves. Over the years they have been getting more and more tools which used to only be available in dedicated Static Analyzers and Lint tools.

Variable shadowing and format-string mismatches can be detected reliably and quickly is because both gcc and clang do this detection as part of their regular compile. – [Bruce Dawson](#)

Here we see two issues (which used to be) very common in C being detected by the two most popular C compilers themselves.

Compiling code with gcc “-Wall -Wextra -pedantic” options catches quite a number of potential or actual problems (<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>). Other compilers check different things as well. So using multiple compilers with their warnings can find plenty of different types of issues for you.

12.7.2 Compiler warnings should be turned in errors on CI.

By getting your errors down to zero on Continuous Integration there is less chance of new warnings being introduced that are missed in code review. There are problems with distributing your code with warnings turned into errors, so that should not be done.

Some points for people implementing this:

- -Werror can be used to turn warnings into errors
- -Wno-error=unknown-pragmas
- should run only in CI, and not in the build by default. See [werror-is-not-your-friend](https://embeddedartistry.com/blog/2017/5/3/-werror-is-not-your-friend) (<https://embeddedartistry.com/blog/2017/5/3/-werror-is-not-your-friend>).
- Use most recent gcc, and most recent clang (change two travis linux builders to do this).
- first have to fix all the warnings (and hopefully not break something in the process).
- consider adding extra warnings to gcc: “-Wall -Wextra -Wpedantic” See [C tooling](#)
- Also the Microsoft compiler MSVC on Appveyor can be configured to treat warnings as errors. The /WX argument option treats all warnings as errors. See [MSVC warning levels](#)
- For MSVC on Appveyor, /wdnnnn Suppresses the compiler warning that is specified by nnnn. For example, /wd4326 suppresses compiler warning C4326.

If you run your code on different CPU architectures, these compilers can find even more issues. For example 32bit/64bit Big Endian, and Little Endian.

12.8 Static analysis tool overview.

Static analysis can be much slower than the analysis usually provided by compilation with gcc and clang. It trades off more CPU time for (hopefully) better results.

[cppcheck](#) focuses of low false positives and can find many actual problems.

[Coverity](#), a commercial static analyzer, free for open source developers [CppDepend](#), a commercial static analyzer based on Clang [codechecker](#), <https://github.com/Ericsson/codechecker>

[cpplint](#), Cpplint is a command-line tool to check C/C++ files for style issues following [Google’s C++ style guide](#).

[Awesome static analysis](#), a page full of static analysis tools for C/C++.

<https://github.com/mre/awesome-static-analysis#cc>

[PVS-Studio](#), a commercial static analyzer, free for open source developers.

12.8.1 The Clang Static Analyzer

The Clang Static Analyzer (<http://clang-analyzer.llvm.org/>) is a free to use static analyzer that is quite high quality.

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. Currently it can be run either as a standalone tool or within Apple Xcode. The standalone tool is invoked from the command line, and is intended to be run in tandem with a build of a codebase.

The talk “Clang Static Analysis” (<https://www.youtube.com/watch?v=UcxF6CVueDM>) talks about an LLVM tool called codechecker (<https://github.com/Ericsson/codechecker>).

On MacOS an up to date scan-build and scan-view is included with the `brew install llvm`.

```
$SCANBUILD=`ls /usr/local/Cellar/llvm/*/bin/scan-build`  
$SCANBUILD -V python3 setup.py build
```

On Ubuntu you can install scan-view with `apt-get install clang-tools`.

12.8.2 cppcheck

Cppcheck is an analysis tool for C/C++ code. It provides unique code analysis to detect bugs and focuses on detecting undefined behaviour and dangerous coding constructs. The goal is to detect only real errors in the code (i.e. have very few false positives).

The quote below was particularly interesting to me because it echos the sentiments of other developers, that testing will find more bugs. But here is one of the static analysis tools saying so as well.

“You will find more bugs in your software by testing your software carefully, than by using Cppcheck.”

To Install cppcheck: <http://cppcheck.sourceforge.net/> and <https://github.com/danmar/cppcheck>

The manual can be found here: <http://cppcheck.net/manual.pdf>

```
brew install cppcheck bear  
sudo apt-get install cppcheck bear
```

To run cppcheck on C code: You can use `bear` (the build ear) tool to record a compilation database (compile_commands.json). cppcheck can then know what c files and header files you are using.

```
# call your build tool, like `bear make` to record.  
# See cppcheck manual for other C environments including Visual Studio.  
bear python setup.py build  
cppcheck --quiet --language=c --enable=all -D__x86_64__ -D__LP64__ --  
→project=compile_commands.json
```

It does seem to find some errors, and style improvements that other tools do not suggest. Note that you can control the level of issues found to errors, to portability and style issues plus more. See `cppcheck --help` and the manual for more details about `--enable` options.

For example these ones from the pygame code base:

```
[src_c/math.c:1134]: (style) The function 'vector_getw' is never used.
[src_c/base.c:1309]: (error) Pointer addition with NULL pointer.
[src_c/scrap_qnx.c:109]: (portability) Assigning a pointer to an integer is not
↳portable.
[src_c/surface.c:832] -> [src_c/surface.c:819]: (warning) Either the condition '!
↳surf' is redundant or there is possible null pointer dereference: surf.
```

12.8.3 /Analyze in Microsoft Visual Studio

Visual studio by Microsoft can do static code analysis too. (<https://docs.microsoft.com/en-us/visualstudio/code-quality/code-analysis-for-c-cpp-overview?view=vs-2017>)
 “Using SAL annotations to reduce code defects.” (<https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects?view=vs-2019>)

“In GNU C and C++, you can use function attributes to specify certain function properties that may help the compiler optimize calls or check code more carefully for correctness.”

<https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html>

12.8.4 Custom static analysis for API usage

Probably one of the most useful parts of static analysis is being able to write your own checks. This allows you to do checks specific to your code base in which general checks will not work. One example of this is the `gcc cpychecker` (<https://gcc-python-plugin.readthedocs.io/en/latest/cpychecker.html>). With this, gcc can find API usage issues within CPython extensions written in C. Including reference counting bugs, and NULL pointer de-references, and other types of issues. You can write custom checkers with LLVM as well in the “Checker Developer Manual” (https://clang-analyzer.lvm.org/checker_dev_manual.html)

There is a list of GCC plugins (<https://gcc.gnu.org/wiki/plugins>) among them are some Linux security plugins by `grsecurity`.

12.9 Runtime checks and Dynamic Verification

Dynamic verification tools examine code whilst it is running. By running your code under these dynamic verification tools you can help detect bugs. Either by testing manually, or by running your automated tests under the watchful eye of these tools. Runtime dynamic verification tools can detect certain errors that static analysis tools can't.

Some of these tools are quite easy to add to a build in Continuous Integration(CI). So you can run your automated tests with some extra dynamic runtime verification enabled.

Take a look at how easy they are to use?

```
./configure CFLAGS="-fsanitize=address,undefined -g" LDFLAGS="-  
↪fsanitize=address,undefined"  
make  
make check
```

12.9.1 Address Sanitizer

Doing a test run with an address sanitizer apparently helps to detect various types of bugs.

AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (runtime flag `ASAN_OPTIONS=detect_stack_use_after_return=1`)
- Use-after-scope (clang flag `-fsanitize-address-use-after-scope`)
- Double-free, invalid free
- Memory leaks (experimental)

How to compile a python C extension with clang on MacOS:

```
LDFLAGS="-g -fsanitize=address" CFLAGS="-g -fsanitize=address -fno-omit-frame-  
↪pointer" python3 setup.py install
```

12.9.2 Undefined Behaviour Sanitizer

From <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>

UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector. UBSan modifies the program at compile-time to catch various kinds of undefined behavior during program execution, for example:

- Using misaligned or null pointer
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination

You can use the Undefined Behaviour Sanitizer with clang and gcc. Here is the [gcc documentation for Instrumentation Options and UBSAN](https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html) (<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>).

From <https://www.sqlite.org/testing.html>

To help ensure that SQLite does not make use of undefined or implementation defined behavior, the test suites are rerun using instrumented builds that try to detect undefined behavior. For example, test suites are run using the “-ftrapv” option of GCC. And they are run again using the “-fsanitize=undefined” option on Clang. And again using the “/RTC1” option in MSVC

To compile a python C extension with a UBSAN with clang on Mac do:

```
LDFLAGS="-g -fsanitize=undefined" CFLAGS="-g -fsanitize=undefined -fno-omit-frame-  
↪pointer" python3 setup.py install
```

12.9.3 Microsoft Visual Studio Runtime Error Checks

The Microsoft Visual Studio compiler can use the Run Time Error Checks feature to find some issues. /RTC (Run-Time Error Checks)

(<https://docs.microsoft.com/en-us/cpp/build/reference/rtc-run-time-error-checks?view=vs-2019>)

From [How to: Use Native Run-Time Checks](#)

(<https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-native-run-time-checks?view=vs-2019>)

- Stack pointer corruption.
- Overruns of local arrays.
- Stack corruption.
- Dependencies on uninitialized local variables.
- Loss of data on an assignment to a shorter variable.

12.9.4 App Verifier

“Any Windows developers that are listening to this: **if you’re not using App Verifier, you are making a mistake.**” – Bruce Dawson

Stangely App Verifier does not have very good online documentation. The best article available online about it is: [Increased Reliability Through More Crashes](#)

(<https://randomascii.wordpress.com/2011/12/07/increased-reliability-through-more-crashes/>)

Application Verifier (AppVerif.exe) is a *dynamic verification* tool for user-mode applications. This tool monitors application actions while the application runs, subjects the application to a variety of stresses and tests, and generates a report about potential errors in application execution or design.

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/application-verifier>

<https://docs.microsoft.com/en-us/windows/win32/win7appqual/application-verifier>

<https://docs.microsoft.com/en-us/security-risk-detection/concepts/application-verifier>

- Buffer over runs
- Use after free issues
- Thread issues including using TLS properly.
- Low resource handling
- Race conditions

If you have a memory corruption bug, App Verifier might be able to help you find it. If you are using Windows APIs wrong, have some threading issues, want to make sure you app runs under harsh conditions – App Verifier might help you find it.

Related to App Verifier is the [PageHeap tool](https://support.microsoft.com/en-us/help/286470/how-to-use-pageheap-exe-in-windows-xp-windows-2000-and-windows-server)(<https://support.microsoft.com/en-us/help/286470/how-to-use-pageheap-exe-in-windows-xp-windows-2000-and-windows-server>) It helps you find memory heap corruptions on Windows.

12.10 Performance profiling and measurement

“The objective (not always attained) in creating high-performance software is to make the software able to carry out its appointed tasks so rapidly that it responds instantaneously, as far as the user is concerned.” Michael Abrash. “Michael Abrash’s Graphics Programming Black Book.”

Reducing energy usage, and run time requirements of apps can often be a requirement or very necessary. For a mobile or embedded application it can mean the difference of being able to run the program at all. Performance can directly be related to user happiness but also to the financial performance of a piece of software.

But how to we measure the performance of a program, and how to we know what parts of a program need improvement? Tooling can help.

12.10.1 Valgrind

Valgrind has its own section here because it does lots of different things for us. It’s a great tool, or set of tools for improving your programs. It used to be available only on linux, but is now also available on MacOS.

Apparently Valgrind would have caught the heartbleed issue if it was used with a fuzzer.

<http://valgrind.org/docs/manual/quick-start.html>

12.10.2 Apple Performance Tools

Apple provides many [performance related development tools](#). Along with the gcc and llvm based tools, the main tool is called [Instruments](#). Instruments (part of Xcode) allows you to record and analyse programs for lots of different aspects of performance - including graphics, memory activity, file system, energy and other program events. By being able to record and analyse different types of events together can make it convenient to find performance issues.

12.10.3 LLVM performance tools.

Many of the low level parts of the tools in XCode are made open source through the LLVM project. See “LLVM Machine Code Analyzer” (<https://llvm.org/docs/CommandGuide/llvm-mca.html>) as one example. See the [LLVM XRay instrumentation](https://llvm.org/docs/XRayExample.html) (<https://llvm.org/docs/XRayExample.html>).

There’s also an interesting talk on XRay here “XRay in LLVM: Function Call Tracing and Analysis” (https://www.youtube.com/watch?v=jjL-__zOGcU) by *Dean Michael Berris*.

12.10.4 GNU/Linux tools

12.10.5 Microsoft performance tools.

12.10.6 Intel performance tools.

<https://software.intel.com/en-us/vtune>

12.11 Caching builds

<https://ccache.samba.org/>

ccache is very useful for reducing the compile time of large C projects. Especially when you are doing a 'rebuild from scratch'. This is because ccache can cache the compilation of parts in this situation when the files do not change.

<http://itscompiling.eu/2017/02/19/speed-cpp-compilation-compiler-cache/>

This is also useful for speeding up CI builds, and especially when large parts of the code base rarely change.

12.12 Distributed building.

distcc <https://github.com/distcc/distcc>

icecream <https://github.com/icecc/icecream>

12.13 Complexity of code.

“Complex is better than complicated. It’s OK to build very complex software, but you don’t have to build it in a complicated way. Lizard is a free open source tool that analyse the complexity of your source code right away supporting many programming languages, without any extra setup. It also does code clone / copy-paste detection.”

Lizard is a modern complexity command line tool,
that also has a website UI: <http://www.lizard.ws/>
<https://github.com/terryyin/lizard>

```
# install lizard python3 -m pip install lizard # show warnings only and include/exclude some files.  
lizard src_c/ -x"src_c/_sdl2*" -w
```

```
# Can also run it as a python module. python3 -m lizard src_c/ -x"src_c/_sdl2*" -w # Show a full  
report, not just warnings (-w). lizard src_c/ -x"src_c/_sdl2*" -x"src_c/_*" -x"src_c/SDL_gfx*" -  
x"src_c/pypm.c"
```

Want people to understand your code? Want Static Analyzers to understand your code better? Want to be able to test your code more easily? Want your code to run faster because of less branches? Then **you may want to find complicated code and refactor it.**

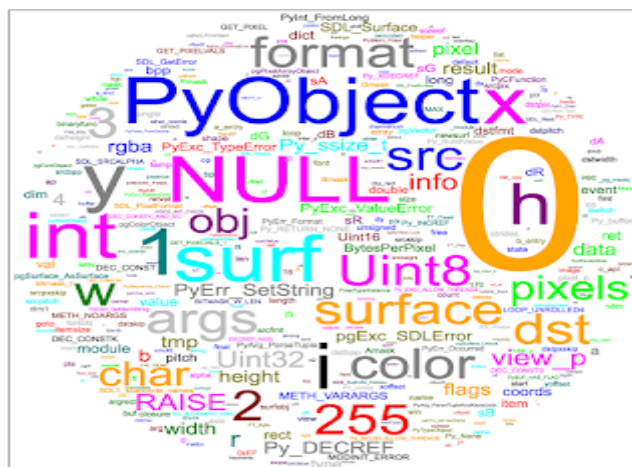


Fig. 3: Lizard can also make a pretty word cloud from your source.

Lizard complexity analysis can be run in Continuous Integration (CI). You can also give it lists of functions to ignore and skip if you can't refactor some function right away. Perhaps you want to stop new complex functions from entering your codebase? To stop new complex functions via CI make a suppression list of all the current warnings and then make your CI use that and fail if there are new warnings.

12.14 Testing your code on different OS/architectures.

Sometimes you need to be able to fix an issue on an OS or architecture that you don't have access to. Luckily these days there are many tools available to quickly use a different system through emulation, or container technology.

Vagrant

Virtualbox

Docker

Launchpad, compile and run tests on many architectures.

Mini cloud (ppc machines for debugging)

If you pay Travis CI, they allow you to connect to the testing host with ssh when a test fails.

12.15 Code Formatting

clang-format

clang-format - rather than manually fix various formatting errors found with a linter, many projects are just using clang-format to format the code into some coding standard.

12.16 Services

LGTM is an ‘automated code review tool’ with github (and other code repos) support.

<https://lgtm.com/help/lgtm/about-automated-code-review>

Coveralls provides a store for test coverage results with github (and other code repos) support.

<https://coveralls.io/>

12.17 Coding standards for C

There are lots of coding standards for C, and there are tools to check them.

An older set of standards is the MISRA_C (https://en.wikipedia.org/wiki/MISRA_C) aims to facilitate code safety, security, and portability for embedded systems.

The Linux Kernel Coding standard (<https://www.kernel.org/doc/html/v4.10/process/coding-style.html>) is well known mainly because of the popularity of the Linux Kernel. But this is mainly concerned with readability.

A newer one is the CERT C coding standard

(<https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>), and it is a secure coding standard (not a safety one).

The website for the CERT C coding standard is quite amazing. It links to tools that can detect each of the problems automatically (when they can be). It is very well researched, and links each problem to other relevant standards, and gives issues priorities. A good video to watch on CERT C is “How Can I Enforce the SEI CERT C Coding Standard Using Static Analysis?” (<https://www.youtube.com/watch?v=awY0iJOkrG4>). They do releases of the website, which is edited as a wiki. At the time of writing the last release into book form was in 2016.

12.18 How are other projects tested?

We can learn a lot by how other C projects are going about their business today.

Also, thanks to CI testing tools defining things in code we can see how automated tests are run on services like Travis CI and Appveyor.

12.18.1 SQLite

“How SQLite Is Tested”

12.18.2 Curl

“Testing Curl”

<https://github.com/curl/curl/blob/master/.travis.yml>

12.18.3 Python

“How is CPython tested?”

<https://github.com/python/cpython/blob/master/.travis.yml>

12.18.4 OpenSSL

“How is OpenSSL tested?”

<https://github.com/openssl/openssl/blob/master/.travis.yml>

They use Coverity too: <https://github.com/openssl/openssl/pull/9805>
<https://github.com/openssl/openssl/blob/master/fuzz/README.md>

12.18.5 libSDL

“How is SDL tested?” [No response]

12.18.6 Linux

<https://stackoverflow.com/questions/3177338/how-is-the-linux-kernel-tested><https://www.linuxjournal.com/content/linux-kernel-testing-and-debugging>

12.18.7 Haproxy

<https://github.com/haproxy/haproxy/blob/master/.travis.yml>

There is some discussion of Post Modern C Tooling on the “C_Programming” reddit forum.

DRAG AND DROP OF FILES WITH PYTHON AND PYGAME

pygame 2 has very basic support for dragging and dropping files.

It receives events “DROPFILE” events with a “file” attribute containing the path to the file dropped. If you drop multiple files on the app it gets multiple “DROPFILE” events.

If you are familiar with pygame events it’s just like this.

```
if e.type == pg.DROPFILE:
    print (e.file) # filename
```

Here’s a very basic example that allows you to drop images and sounds on it.

- draws images dropped on it
- plays sounds dropped on it

```
#!/usr/bin/env python
import os.path
import pygame as pg

def main():
    """ Very simple example of using the DROPFILE event type
        to handle dragging and dropping of files.

        If an image file is dropped from the file manager we load it,
        and display it in the window.
    """
    going = True
    pg.init()
    screen = pg.display.set_mode((640, 480))
    pg.display.set_caption('Drag an image file onto the window.')
    clock = pg.time.Clock()
    image = None

    while going:
        for e in pg.event.get():
            if e.type == pg.QUIT or (e.type == pg.KEYDOWN and e.key == pg.K_
↳ESCAPE):
                going = False
            elif e.type == pg.DROPFILE:
                file_extension = os.path.splitext(e.file)[1]
                if file_extension in [".bmp", ".jpg", ".png"]:
                    try:
                        image = pg.image.load(e.file) # load the image given the_
↳file name.

                        pg.display.set_caption(e.file) # show file name in title_
↳bar.

                        screen.blit(image, (0, 0))
                        pg.display.flip()
```

(continues on next page)

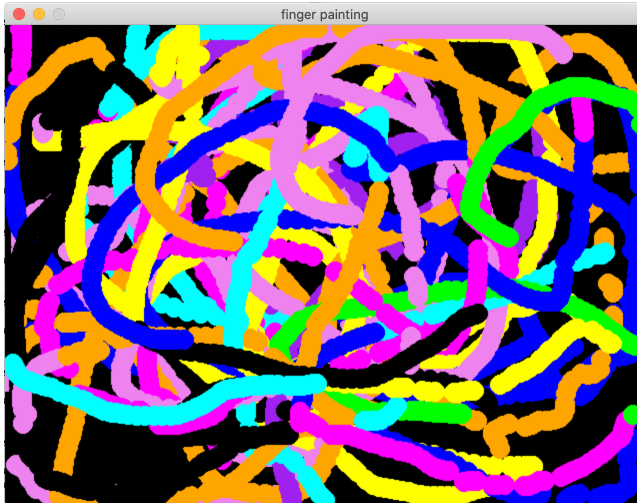
(continued from previous page)

```
        except:
            pass
    if file_extension in [".mp3", ".ogg", ".wav"]:
        try:
            pygame.mixer.music.load(e.file) # load the music in the_
↪file name.
            pygame.mixer.music.play()
        except:
            pass

    clock.tick(30)

if __name__ == "__main__":
    main()
```

FINGER PAINTING WITH MULTI-TOUCH



Touch screens like on phones, and some track pads on laptops allow tracking more than one point of contact. New in pygame 2 is support for Multi-Touch.

In this piece:

- We will describe some new finger tracking events,
- show a demo of how to use them in a “finger painting” program.

14.1 Finger events

pygame 2 brings these new event types.

FINGERDOWN A finger has touched the surface.

FINGERMOTION A finger is moving on the surface.

FINGERUP A finger is not touching the surface anymore.

14.1.1 Finger event attributes

Each of the types of events has the same attributes.

touch_id (int) the touch device id. So you can see which device this is from. This is useful if you have say a multi touch screen, and a touch pad.

finger_id (int) the finger id Lots of people have 10 fingers. Multi touch devices can often detect a number of different fingers moving around at once.

x (float) the x-axis location of the touch event, normalized (0...1)

y (float) the y-axis location of the touch event, normalized (0...1)

dx (float) the distance moved in the x-axis, normalized (-1...1)

dy (float) the distance moved in the y-axis, normalized (-1...1)

pressure (float) the quantity of pressure applied, normalized (0...1)

14.2 Code: finger_painting_multi_touch.py

Here we have a nice little demo of using the FINGER events.

```
#!/usr/bin/env python
import os
import pygame as pg

# This makes the touchpad be usable as a multi touch device.
os.environ['SDL_MOUSE_TOUCH_EVENTS'] = '1'

def main():
    pg.init()

    # Different colors for different fingers.
    colors = [
        'red', 'green', 'blue', 'cyan', 'magenta',
        'yellow', 'black', 'orange', 'purple', 'violet'
    ]
    available_colors = colors[:] + colors[:] # two copies for people with 12_
    ↪ fingers.

    # keyed by finger_id, and having dict as a value like this:
    # {
    #     'x': 20,
    #     'y': 20,
    #     'color': 'red',
    # }
    circles = {}

    width, height = (640, 480)
    screen = pg.display.set_mode((width, height))
    clock = pg.time.Clock()
    pg.display.set_caption('finger painting with multi-touch')
    # we hide the mouse cursor and keep it inside the window.
    pg.event.set_grab(True)
    pg.mouse.set_visible(False)

    going = True
    while going:
        for e in pg.event.get():

            # We look for finger down, finger motion, and then finger up.
            if e.type == pg.FINGERDOWN:
                circles[e.finger_id] = {
                    'color': available_colors.pop(),
```

(continues on next page)

(continued from previous page)

```

        'x': int(width * e.x), # x and y are 0.0 to 1.0 in touch_
↪space.
        'y': int(height * e.y), # we translate to the screen_
↪pixels.
    }
    elif e.type == pg.FINGERMOTION:
        circles[e.finger_id].update({
            'x': int(width * e.x),
            'y': int(height * e.y),
        })
    elif e.type == pg.FINGERUP:
        available_colors.append(circles[e.finger_id]['color'])
        del circles[e.finger_id]
    elif ((e.type == pg.KEYDOWN and e.key in (pg.K_q, pg.K_ESCAPE))
or e.type == pg.QUIT):
        going = False

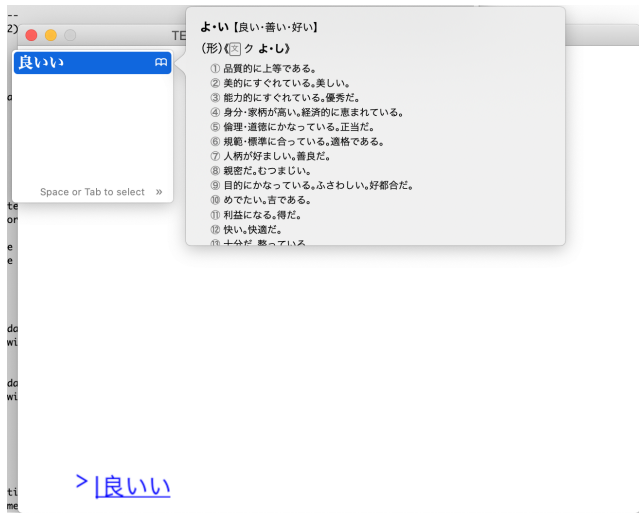
    # lets draw a circle for each finger.
    for finger_id, circle in circles.items():
        pg.draw.circle(screen, circle['color'], (circle['x'], circle['y']), 10)

    clock.tick(60)
    pg.display.flip()

if __name__ == "__main__":
    main()

```


TEXT EDITING INPUT IME



Why is TEXTINPUT needed when we have KEYDOWN events? It's because one key press doesn't always match up to one character generated. Sometimes several characters can be generated. This is why the old KEYDOWN events are not sometimes enough.

Input Method Editor (IME) is an operating system function that lets you generate different characters that may even be on your input device. Especially popular with languages such as Korean, Chinese, Japanese, Vietnamese, and Indic characters - but is also used in European languages

15.1 New events, their fields, and description

15.1.1 pygame.TEXTINPUT

- type: int, pygame.TEXTINPUT
- text: str, the UTF-8 editing text.

15.1.2 pygame.TEXTEDITING

- type: int, pygame.TEXTEDITING
- text: str, the UTF-8 editing text.
- start: int, the location to begin editing from

- length: int, the number of characters to edit from the start point

If you are familiar with pygame events it's just like this.

```
# We need to enable text input to get the IME going, and to get TEXTEDITING events.
pygame.key.start_text_input()

if e.type == pg.TEXTEDITING:
    print(e.text)    # the text being edited
    print(e.start)   # start of text editing
    print(e.length)  # selection length

    if e.type == pg.TEXTINPUT:
        print(e.text) # the text input after editing is done.
```

15.2 Try IME yourself?

Maybe you want to try it out, but usually you use basic English?

- Mac: How to set up Japanese on Mac. <https://support.apple.com/guide/japanese-input-method/set-up-the-input-source-jpim10267/mac>
- Windows: “Installing and Using Input Method Editors” <https://docs.microsoft.com/en-us/windows/win32/dxtechcharts/installing-and-using-input-method-editors>
- Ubuntu: “Japanese Input This article provides information on how to setup Japanese input on an English Ubuntu installation.” <https://help.ubuntu.com/community/JapaneseInput>

15.3 Larger example

```
#!/usr/bin/env python
""" A mini chat app.

A little "console" where you can write in text.

Shows how to use the TEXTEDITING and TEXTINPUT events.

pygame.key.start_text_input() - Start receiving TEXTEDITING and TEXTINPUT events.
pygame.key.stop_text_input() - Stop receiving TEXTEDITING and TEXTINPUT events.
pygame.key.set_text_input_rect() - rectangle may be used for IME where candidate list
↪may open.
"""
from collections import deque
import pygame as pg
import pygame.freetype as freetype

def load_fonts():
    fontnames = ",".join(
        (
            "hiraginosansgb", # japanese
            "hiraginosanscns", # japanese
            "Arial",
        )
    )
    font = freetype.SysFont(fontnames, 24)
    font_small = freetype.SysFont(fontnames, 16)
    return font, font_small
```

(continues on next page)

(continued from previous page)

```

class EditingText:
    """A state machine which handles text editing events."""

    def __init__(self, chat_entries):
        self.pos = 0
        self.editing = False
        self.text = ""
        self.text_pos = 0
        self.editing_text = ""
        self.editing_pos = 0
        self.chat_entries = chat_entries

    def state(self):
        return (
            self.pos,
            self.editing,
            self.text,
            self.text_pos,
            self.editing_text,
            self.editing_pos,
        )

    def reset(self):
        self.text = ""
        self.text_pos = 0

    def text_editing(self, start, text):
        self.editing = True
        self.editing_text = text
        self.editing_pos = start

    def text_input(self, text):
        self.editing = False
        self.editing_text = ""
        self.text = self.text[0 : self.text_pos] + text + self.text[self.text_pos_]
↪:]
        self.text_pos += len(text)

    def backspace(self):
        if len(self.text) > 0 and self.text_pos > 0:
            self.text = self.text[0 : self.text_pos - 1] + self.text[self.text_pos_]
↪:]
            self.text_pos = max(0, self.text_pos - 1)

    def delete(self):
        self.text = self.text[0 : self.text_pos] + self.text[self.text_pos + 1 :]

    def left(self):
        self.text_pos = max(0, self.text_pos - 1)

    def right(self):
        self.text_pos = min(len(self.text), self.text_pos + 1)

    def enter(self):
        if self.text:
            self.chat_entries.append(self.text)
            self.reset()

```

(continues on next page)

(continued from previous page)

```

class HandleEditing(EditingText):
    """ This keeps the pygame event handling separate from EditingText.
    """
    def handle_events(self, events):
        for event in events:
            if event.type == pg.TEXTEDITING:
                self.text_editing(start=event.start, text=event.text)
            elif event.type == pg.TEXTINPUT:
                self.text_input(event.text)
            elif event.type == pg.KEYDOWN:
                if self.editing:
                    if len(self.editing_text) == 0:
                        self.editing = False
                        continue
                if event.key == pg.K_BACKSPACE:
                    self.backspace()
                elif event.key == pg.K_DELETE:
                    self.delete()
                elif event.key == pg.K_LEFT:
                    self.left()
                elif event.key == pg.K_RIGHT:
                    self.right()
                elif (
                    event.key in [pg.K_RETURN, pg.K_KP_ENTER]
                    and len(event.unicode) == 0
                ):
                    self.enter()
        return events

class StateSprite(pg.sprite.DirtySprite):
    """ This makes it easier to track state changes,
    and only redraw if things change.
    """
    _last_state = None
    dirty = 0
    def check_dirty(self):
        state = self.state()
        if state == self._last_state:
            self.dirty = 0
            return False
        self._last_state = state
        self.dirty = 1
        return True

class ChatInputEdit(StateSprite):
    def __init__(self, editing, font, text_color, background_color):
        StateSprite.__init__(self)
        self.editing = editing
        self.font = font
        self.text_color = text_color
        self.background_color = background_color
        self.rect = pg.Rect(30, 440, pg.display.get_surface().get_width(), 40)
        self.image = pg.Surface((self.rect.width, self.rect.height)).convert_
        ↪alpha()

    def state(self):
        return self.editing.state()

    def update(self):
        if not self.check_dirty():

```

(continues on next page)

(continued from previous page)

```

        return

    editing = self.editing
    font = self.font
    text_color = self.text_color

    self.image.fill(self.background_color)

    start_pos = pg.Rect(30, 0, self.rect.w, self.rect.h)
    ime_text_l = "> " + editing.text[0 : editing.text_pos]
    ime_text_m = (
        editing.editing_text[0 : editing.editing_pos]
        + "|"
        + editing.editing_text[editing.editing_pos :]
    )
    ime_text_r = editing.text[editing.text_pos :]

    rect_text_l = font.render_to(self.image, start_pos, ime_text_l, text_color)
    start_pos.x += rect_text_l.width

    rect_text_m = font.render_to(
        self.image, start_pos, ime_text_m, text_color, None, freetype.STYLE_
↳UNDERLINE
    )
    start_pos.x += rect_text_m.width
    font.render_to(self.image, start_pos, ime_text_r, text_color)

class ChatList(StateSprite):
    def __init__(self, chat_entries, font, text_color, background_color):
        StateSprite.__init__(self)
        self.chat_entries = chat_entries
        self.font = font
        self.text_color = text_color
        self.background_color = background_color
        self.rect = pg.Rect(30, 20, pg.display.get_surface().get_width(), 400)
        self.image = pg.Surface((self.rect.width, self.rect.height)).convert_
↳alpha()

    def state(self):
        return self.chat_entries.copy()

    def update(self):
        if not self.check_dirty():
            return
        chat_entries = self.chat_entries
        font = self.font
        text_color = self.text_color

        self.image.fill(self.background_color)
        chat_entries_pos = self.rect
        chat_height = chat_entries_pos.height / chat_entries.maxlen
        for i, chat in enumerate(chat_entries):
            font.render_to(
                self.image,
                (chat_entries_pos.x, chat_entries_pos.y + i * chat_height),
                chat,
                text_color,
            )

```

(continues on next page)

(continued from previous page)

```
def main():
    text_color = "blue"
    background_color = "white"

    pg.init()
    screen = pg.display.set_mode((640, 480))
    screen.fill(background_color)

    pg.display.set_caption("TEXTINPUT chat example. Loading...")
    pg.event.pump()

    clock = pg.time.Clock()
    font, font_small = load_fonts()
    pg.display.set_caption("TEXTINPUT chat example. Enter your chat messages.")

    pg.key.start_text_input()
    pg.key.set_text_input_rect(pg.Rect(80, 80, 320, 40))

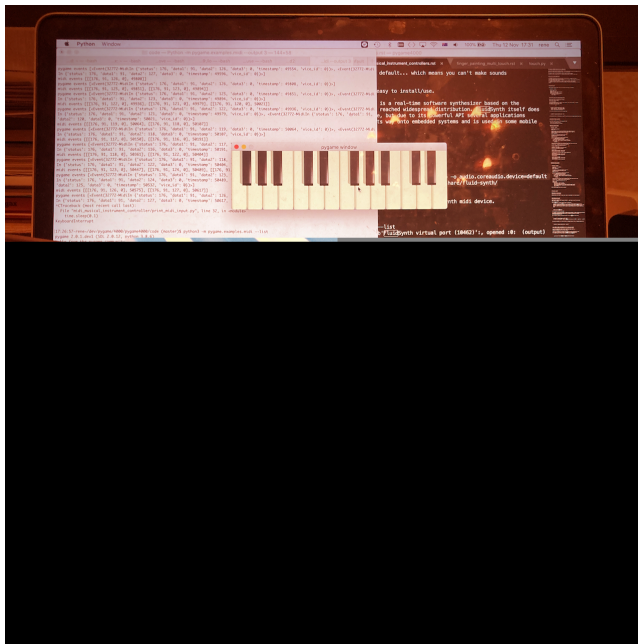
    editing = HandleEditing(chat_entries=deque([], maxlen=20))
    allsprites = pg.sprite.LayeredDirty((
        ChatList(editing.chat_entries, font_small, text_color, background_color),
        ChatInputEdit(editing, font, text_color, background_color),
    ))

    background = screen.convert()
    background.fill(background_color)
    allsprites.clear(screen, background)

    going = True
    while going:
        events = pg.event.get()
        editing.handle_events(events)
        if [e for e in events if e.type == pg.QUIT]:
            going = False
        allsprites.update()
        rects = allsprites.draw(screen)
        pg.display.update(rects)
        clock.tick(50)

if __name__ == "__main__":
    main()
```

MIDI MUSICAL INSTRUMENT CONTROLLERS AND SYNTHESIZERS



Musical Instrument Digital Interface (Midi), is for letting digital musical instruments talk to each other. It came from the early 80s, and is still used today.

The main idea is that rather than sending audio data, it sends small control messages. This is great for fast real time low bandwidth communication.

I'm going to show in all the different ways midi can be used with pygame, and how to set up a virtual synth (I've included instructions for Win, Mac, and Linux).

16.1 What can you do with pygame.midi?

- 1) **Play:** If you want to play a midi file, then you can use `pygame.mixer.music`.
- 2) **List:** See what midi devices are available on your computer. Maybe connected via USB or via a sound card with midi cables.
- 3) **Input:** Use music controllers as input for your app (like a midi keyboard).
- 4) **Output:** Send midi events to musical instruments (like analog synthesizers, or virtual synths)

16.2 Playing midi files

pygame supports playing midi files so you can hear the music via a speaker.

```
import pygame, time
pygame.mixer.init()
pygame.mixer.music.load('file.midi')
pygame.mixer.music.play()
while pygame.mixer.music.get_busy():
    print('Still playing :')
    time.sleep(1)
```

16.3 Listing midi devices with pygame.examples.midi

pygame comes with an interesting midi example. It has three features really.

- 1) List midi devices
- 2) Use an onscreen keyboard to output to midi devices (like virtual synths, or musical instruments at the end of midi cables)
- 3) Print the midi input events

```
$ python3 -m pygame.examples.midi --help
--input [device_id] : Midi message logger
--output [device_id] : Midi piano keyboard
--list : list available midi devices
```

To list the midi devices you can use the example app.

```
$ python3 -m pygame.examples.midi --list
```

See the section later on “Setting up a synth” to see how to use the pygame.examples.midi more.

16.4 pygame.midi API basic explainer

Now we will show the basics of this pygame.midi API.

- How to list the output devices.
- How to send midi output to a device.

16.4.1 pygame.midi.Output example

```
# code/midi_musical_instrument_controller/midi_play_note.py
import time

# import pygame.midi and initialize the midi module before you use it.
# pygame.midi is not a default pygame module so you need to import it,
# and also pygame.init does not init it for you.
import pygame.midi

# initialize the midi module before you use it.
# pygame.init does not do this for you.
pygame.midi.init()

# print the devices and use the last output port.
```

(continues on next page)

(continued from previous page)

```

for i in range(pygame.midi.get_count()):
    r = pygame.midi.get_device_info(i)
    (interf, name, is_input, is_output, is_opened) = r
    print (interf, name, is_input, is_output, is_opened)
    if is_output:
        last_port = i

# You could also use this to use the default port rather than the last one.
# default_port = pygame.midi.get_default_output_id()

midi_out = pygame.midi.Output(last_port, 0)

# select an instrument.
instrument = 19 # general midi church organ.
midi_out.set_instrument(instrument)

# play a note.
midi_out.note_on(note=62, velocity=127)
midi_out.note_off(note=62, velocity=0)

# sleep for a bit, and play another higher pitched note.
time.sleep(0.2)
midi_out.note_on(note=80, velocity=127)
midi_out.note_off(note=80, velocity=0)
time.sleep(0.2)

# play a note for longer.
midi_out.note_on(note=62, velocity=127)
time.sleep(0.8)
midi_out.note_off(note=62, velocity=0)

```

16.4.2 pygame.midi.Input example

Here we see how to read midi events from a device like an attached USB keyboard.

```

# code/midi_musical_instrument_controller/print_midi_input.py
import time
import pygame.midi

# initialize the midi module before you use it.
pygame.midi.init()

# print the devices
for i in range(pygame.midi.get_count()):
    r = pygame.midi.get_device_info(i)
    (interf, name, is_input, is_output, is_opened) = r
    print (interf, name, is_input, is_output, is_opened)

device_id = pygame.midi.get_default_input_id()

print('Using device id: %s' % device_id)
midi_in = pygame.midi.Input(device_id)
going = True
print ('Use ctrl+c to quit')
while going:
    midi_events = midi_in.read(20)
    if midi_events:
        print('midi events', midi_events)
        # make some pygame.Events to be used like other pygame events.

```

(continues on next page)

(continued from previous page)

```
print('pygame events', pygame.midi.midis2events(midi_events, device_id))
time.sleep(0.1)
```

16.5 Setting up a synth

16.5.1 Windows built in midi synth

Windows comes with a built in midi synth, which is detectable as an output device.

16.5.2 MacOS Setting up a synth

MacOS doesn't come with one running by default... which means you can't make sounds with your midi Output.

Luckily fluidsynth is pretty good and easy to install/use.

A SoundFont Synthesizer FluidSynth is a real-time software synthesizer based on the SoundFont 2 specifications and has reached widespread distribution. FluidSynth itself does not have a graphical user interface, but due to its powerful API several applications utilize it and it has even found its way onto embedded systems and is used in some mobile apps.

```
$ brew install fluid-synth
```

In one terminal tab start the synth:

```
$ fluidsynth -o midi.driver=coremidi -o audio.driver=coreaudio -o audio.coreaudio.device=default -o
audio.period-size=256 /usr/local/Cellar/fluid-synth/2.1.2/share/fluid-synth/sf2/VintageDreamsWaves-v2.sf2
```

Then in another terminal you should be able to see the FluidSynth midi device.

```
$ python3 -m pygame.examples.midi --list
0: interface :b'CoreMIDI':, name :b'FluidSynth virtual port (10462)':, opened :0:  ↳
↳ (output)
```

Now play some sounds on it using the built in keyboard which is part of pygame.examples.midi

```
$ python3 -m pygame.examples.midi --output 0
```

16.5.3 Linux: Debian, Ubuntu and Raspberry Pi OS - setup a midi synth

Maybe you already have a midi synth in your linux setup, but it is becoming increasingly rare. So, just in case...

There are an explosion of different audio and midi systems available on linux. We are going to setup both timidity and fluidsynth virtual synthesizers. If you were going to use only one, then perhaps choose fluidsynth.

```
sudo apt-get install fluidsynth fluid-soundfont-gm timidity
```

Fluidsynth

```
$ fluidsynth -a alsa -m alsa_seq /usr/share/sounds/sf2/FluidR3_GM.sf2

$ python3 -m pygame.examples.midi --list
0: interface :b'ALSA':, name :b'Midi Through Port-0':, opened :0:  (output)
1: interface :b'ALSA':, name :b'Midi Through Port-0':, opened :0:  (input)
2: interface :b'ALSA':, name :b'TiMidity port 0':, opened :0:  (output)
```

(continues on next page)

(continued from previous page)

```

3: interface :b'ALSA':, name :b'TiMidity port 1':, opened :0: (output)
4: interface :b'ALSA':, name :b'TiMidity port 2':, opened :0: (output)
5: interface :b'ALSA':, name :b'TiMidity port 3':, opened :0: (output)
6: interface :b'ALSA':, name :b'Synth input port (5852:0)':, opened :0: (output)

$ python3 -m pygame.examples.midi --output 6

```

Timidity

In one shell run the timidity server.

```
timidity -iA
```

Now we should be able to see a whole lot of devices listed.

```

$ python -m pygame.examples.midi --list
0: interface :ALSA:, name :Midi Through Port-0:, opened :0: (output)
1: interface :ALSA:, name :Midi Through Port-0:, opened :0: (input)
2: interface :ALSA:, name :TiMidity port 0:, opened :0: (output)
3: interface :ALSA:, name :TiMidity port 1:, opened :0: (output)
4: interface :ALSA:, name :TiMidity port 2:, opened :0: (output)
5: interface :ALSA:, name :TiMidity port 3:, opened :0: (output)
6: interface :ALSA:, name :TiMidity port 0:, opened :0: (output)
7: interface :ALSA:, name :TiMidity port 1:, opened :0: (output)
8: interface :ALSA:, name :TiMidity port 2:, opened :0: (output)
9: interface :ALSA:, name :TiMidity port 3:, opened :0: (output)

```

The following command brings up a mini keyboard to play notes. Try a few ports to see if they make any sound... maybe ports 6 and 7 will work for you too.

```
$ python -m pygame.examples.midi --output 6
```

Trouble shooting on linux

On some linux systems if you install pygame with wheels you may get an error about “Cannot open shared library...” or so.

```

ALSA lib conf.c:3558:(snd_config_hooks_call) Cannot open shared library libasound_
↳module_conf_pulse.so (/usr/lib/alsa-lib/libasound_module_conf_pulse.so:
↳libasound_module_conf_pulse.so: cannot open shared object file: No such file or
↳directory)
ALSA lib seq.c:935:(snd_seq_open_noupdate) Unknown SEQ default

```

There are two workarounds available to this issue.

- You could try installing pygame with apt-get instead.
- You could try symlinking the folder to the one that the pygame wheel alsa is looking for. Which of these symlinks of these to use depends on if your system is i386 or x86_64.

```

sudo ln -s /usr/lib/i386-linux-gnu/alsa-lib /usr/lib/alsa-lib
sudo ln -s /usr/lib/x86_64-linux-gnu/alsa-lib /usr/lib/alsa-lib

```


SOUND GENERATION AND DRAWING

Here's a few sound generation examples with pygame (and no numpy/scipy).

All the basics for making a music program (sampler/synth).

- some sample rate conversion,
- bit rate conversion
- tone generation using generators (square wave)
- python arrays used as buffers for pygame.Sound (zero copy).

code/sound_generation_and_drawing/sound_samples.py

- a simple 'square wave' generated
- resampling sample rates (eg, 8363 to 44100)
- using built in python array for making pygame.Sound samples.
- samples at different bit sizes
- converting from signed 8 to signed 16bit
- how initializing the mixer changes what samples Sound needs.
- Using the python stdlib audioop.ratecv for sample rate conversion.
- drawing sound sample arrays as a waveform scaled into a Surface.

17.1 Metronome.

code/sound_generation_and_drawing/metronome.py

HOW TO PORT AND MARKET GAMES USING #PYTHON AND #PYGAME.

You've spent two years making a game, but now want other people to see it?

How do you port it to different platforms, and make it available to others? How do you let people know it is even a thing? Is your game Free Libre software, or shareware?

All python related applications are welcome on www.pygame.org. You'll need a screenshot, a description of your game, and some sort of URL to link people to (a github/gitlab/bitbucket perhaps). But how and where else can you share it?

18.1 a few platforms to port to

- itch.io and windows
- windows store?
- mac (for itch.io)
- mac store
- steam
- linux 'flatpack' (latest fedora/ubuntu etc use this like an app store).
- pypi (python packages can actually be installed by lots of people)
- android store
- web
- debian
- redhat/fedora

18.2 Make it a python package

Some of the tools work more easily with your package as a python package. Working with all the different tools is sort of hard, and having a convention for packaging would make things easier.

Python packaging guide - <http://packaging.python.org/>

So, why don't we do things as a simple python package with one example app to do this? pygame has an example app already, solarwolf - <https://github.com/pygame/solarwolf>. With work, it could be a good example app to use. We can also link in this guide to other pygame apps that have been distributed on various places.

There are other example apps linked below for different distribution technology.

18.2.1 Where to put your files?

The [stuntcat community game](#) is setup as an example of how you can package your game.

It's not the only way to structure your files, but there are some advantages to doing it this way.

- Follows the python packaging guidelines. It's just a python package.
- Uses free Appveyor and TravisCI accounts to make Windows, Mac, and Linux binaries.
- Is set up to run tests when you push to github.
- Makes a windows, mac, linux, and python package.
- Uses different python packages like pymunk, thorpy, pycscroll.
- It will be kept working, and ported to more and more platforms as time goes on.

So if you're looking for inspiration on how to structure your files, it's a good place to look.

18.2.2 pyinstaller

<https://www.pyinstaller.org/>

This can make ones for linux, windows, and mac.

```
pyinstaller --onefile --windowed --icon=icon.ico .py
```

18.3 Windows

pysist and pyinstaller can be used.

<https://github.com/takluyver/pysist>

The benefit of pysist is that it can create installers. Whereas pyinstaller is for making standalone executables (which is good if you are putting your app on the Steam store for example).

18.3.1 Windows code signing

- <https://github.com/pyinstaller/pyinstaller/wiki/Recipe-Win-Code-Signing>
- <https://docs.microsoft.com/en-us/windows/uwp/packaging/create-certificate-package-signing>

18.4 Flatpak

apps on linux - <https://flatpak.org/>

Here's an example of making a pygame one. <https://github.com/flathub/flathub/pull/478>

Developer guide for more detail here - <http://docs.flatpak.org/en/latest/>

18.5 pypi

The python package system can mean your app can be available for everyone who can use pip. Which is an audience in the millions.

18.6 Mac

pyinstaller is probably the best option at the moment. If your game is open source, then you could use TravisCI for free to make builds with pyinstaller.

Unfortunately you probably need a Mac to make a mac build, test it, and release on the mac/ios stores. Getting a cheap apple machine off ebay might be the way to go. Or a cloud account perhaps from 'macincloud'. Also the mac developer program costs \$100.

Another option might be to borrow a friends machine to make the builds when it's time.

See:

- <https://github.com/pyinstaller/pyinstaller/wiki/Recipe-OSX-Code-Signing>
- <https://developer.apple.com/macos/distribution/>
- <https://developer.apple.com/support/code-signing/>

18.7 iOS

It's not easy, but possible.

With pygame 2 this should be possible since it uses the new SDL2.

If you use LGPL code on iOS you still have to let your users benefit from the protections the LGPL gives them.

Tom from renpy says... "I've been distributing Ren'Py under LGPL section 6c, which says that you can distribute it along with a written offer to provide the source code required to create the executables. Since Ren'Py has a reasonably strong distinction between the engine and game scripts, the user can then combine the game data from an iOS backup with the newly-linked Ren'Py to get a package they can install through xcode."

https://github.com/renpy/pygame_sdl2/issues/109#issuecomment-412156973

An apple developer account costs \$100, and selling things costs 30% of the cost of your app.

<https://developer.apple.com/>

18.8 Steam

There's a few games released using pygame on steam. Here are two threads of games released:

- https://www.reddit.com/r/pygame/comments/87q9fr/i_just_published_my_game_the_four_colour_theorem/
- https://www.reddit.com/r/pygame/comments/4ck5zv/released_a_pygame_game_on_steam_after_3_years_of/
- https://www.reddit.com/r/pygame/comments/3j6zvr/with_help_from_you_guys_my_first_game_launched_on/

Costs \$100 to join up and sell a game on this store. <https://partner.steamgames.com/>

Recently someone used pyinstaller to package thier game.

```
pyinstaller --onefile --windowed --icon=icon.ico .py
```

18.8.1 SteamworksPy

A python module for the C++ steam sdk. <https://github.com/Gramps/SteamworksPy>

Made by someone who has released their game (using pygame) on steam.

18.9 Itch.io

“itch.io is an open marketplace for independent digital creators with a focus on independent video games.”

- creators FAQ - <https://itch.io/docs/creators/faq>

Quite a few people have released their pygame games on itch.io.

18.10 Android

This isn't really possible to do well at the moment without a bit of work.

python-for-android seems the best option, but doesn't work well with pygame.

<https://github.com/kivy/python-for-android> There is an old and unmaintained pygame recipe included (for an old pygame 1.9.1). With some work it should be possible to update the recipe to use the SDL2 support in pygame.

There was an older 'pygame subset for android' which is now unmaintained, and does not work with more recent Android devices.

18.11 Web

There's not really an 'export for web' option at the moment. It is possible with both CPython and SDL as well as SDL2 working on emscripten (the compiler for WASM and stuff that goes on the web).

Here is the latest 'cpython on web' project. <https://github.com/iodide-project/pyodide>

18.12 Building if you do not have a windows/mac/linux machine

18.12.1 CI tools

If your game is open source, you can use these systems to build your game remotely for free.

- Appveyor (for windows) <https://www.appveyor.com/>
- Travis (for linux and mac) <https://travis-ci.org/>

How to do that? Well, that's an exercise left up to the reader. Probably getting it to use pyinstaller, and having them upload the result somewhere.

One python app that uses Travis and Appveyor is the Mu editor. You can see how in their .travis.yml and appveyor.yml files. See <https://github.com/mu-editor/mu>

18.12.2 Virtualbox

With virtualbox (and other emulators) you can run some systems on your local machine. Which means you do not need to buy a new development machine yourself for those platforms.

Both windows and linux images are available that you could use legally.

<https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/>

Note, that it is good to do your testing on a free install, rather than testing on the same machine that you made your executables with. This is because perhaps you forgot to include some dependency, and that dependency is on the development machine, but not everyone else's machines.

18.13 Writing portable python code

Some old (but still valid) basic advice on making your game portable: <https://www.pygame.org/wiki/distributing>
Things like naming your files case sensitively.

18.14 Announcing your game.

Generic Indie game marketing guides all apply here.

Some python/pygame specific avenues for marketing and announcing. . .

- post a 'release' on the pygame website
- mention it on the pygame reddit (and python reddit perhaps)
- announce on the python announce mailing list (<https://mail.python.org/mailman/listinfo/python-announce-list>)
- get your blog listed on planet with your 'python' tag. (open an issue <https://github.com/python/planet/issues>)
- mention @pygame_org or #pygame for retweets

Of course the python world is a tiny place compared to the entire world.

- <https://www.reddit.com/r/gamedevexpo/>

18.15 Icons

Each platform has slightly different requirements for icons. This might be a nice place to link to all the requirements (TODO).

18.16 Making a game trailer (for youtube)

You may not need to make the best trailer, or even a good trailer. Just a screen capture of your game might be 'good enough' and is better than nothing.

How about making a trailer with pygame itself? You could call it 'demo mode', or 'intro mode'.

There's a free iMovie on Mac, the Microsoft video editor on windows, and blender for all platforms. An alternative is to use the python module [moviepy](#) and script your game trailer.

OBS is pretty good multi platform free screen capture software. <https://obsproject.com/download>

- [the top 10 best indie game trailers 2018](#)
- [How to Make an Indie Game Trailer With No Budget](#)

18.17 Animated gif

These are useful for sharing on twitter and other such places, so people can see game play.

You can save the .png files with pygame, and convert them to a gif with the 'convert' tool from imagemagik.

```
# brew install imagemagick
# sudo apt-get install imagemagick

# call this in your main loop.
pygame.image.save(surf, 'bla_%05d.png' % frame_idx)
```

Now you can convert the png files to

```
convert -delay 20 -loop 0 bla_*.png animated.gif
```

Some solutions on stack overflow.

- <https://stackoverflow.com/questions/10922285/is-there-a-simple-way-to-make-and-save-an-animation-with-pygame>
- <https://stackoverflow.com/questions/753190/programmatically-generate-video-or-animated-gif-in-python>

LET'S WRITE A UNIT TEST!

A unit test is a piece of code which tests one thing works well in isolation from other parts of software. In this guide, I'm going to explain how to write one using the standard python unittest module, for the `pygame` game library. You can apply this advice to most python projects, or free/libre open source projects in general.

19.1 A minimal test

What `pygame.draw.ellipse` should do: <http://www.pygame.org/docs/ref/draw.html#pygame.draw.ellipse>

Where to put the test: https://github.com/pygame/pygame/blob/main/test/draw_test.py

```
def test_ellipse(self):
    import pygame.draw
    surf = pygame.Surface((320, 200))
    pygame.draw.ellipse(surf, (255, 0, 0), (10, 10, 25, 20))
```

All the test does is call the draw function on the surface with a color, and a rectangle. That's it. A minimal, useful test. If you have a github account, you can even edit the test file in the browser to submit your PR. If you have email, or internet access you can email me or someone else on the internet and ask them to do add it to pygame. An easy test to write... but it provides really good value.

- Shows an example of using the code.
- Makes sure the function arguments are correct.
- Makes sure the code runs on 20+ different platforms and python versions.
- No "regressions" (Code that starts failing because of a change) can be introduced in the future. The code for draw ellipse with these arguments should not crash in the future.

19.2 But why write a unit test anyway?

Unit tests help pygame make sure things don't break on multiple platforms. When your code is running on dozens of CPUs and just as many operating systems things get a little tricky to test manually. So we write a unit test and let all the build robots do that work for us.

A great way to contribute to libre/free and open source projects is to contribute a test. Less bugs in the library means less bugs in your own code. Additionally, you get some public credit for your contribution. The best part about it, is that it's a great way to learn python, and about the thing you are testing. Want to know how graphics algorithms should work, in lots of detail? Start writing tests for them.

The simplest test is to just call the function. Just calling it is a great first test. Easy, and useful.

At the time of writing there are 39 functions that aren't even called when running the pygame tests. Why not join me on this adventure?

19.3 Let's write a unit test!

In this guide I'm going to write a test for an `pygame.draw.ellipse` to make sure a thick circle has the correct colors in it, and not lots of black spots. There's a bunch of tips and tricks to help you along your way. Whilst you can just edit a test in your web browser, and submit a PR, it might be more comfortable to do it in your normal development environment.

19.3.1 Grab a fork, and let's dig in.

Set up [git](https://github.com) for [github](https://github.com) if you haven't already. Then you'll want to 'fork' pygame on <https://github.com/pygame/pygame> so you have your own local copy.

Note, we also accept patches by email, or on github issues. So you can skip all this github business if you want to. <https://www.pygame.org/wiki/patchesandbugs>

- Fork the repository (see top right of the [pygame repo page](#))



- Make the change locally. Push to your copy of the fork.
- Submit a *pull request*

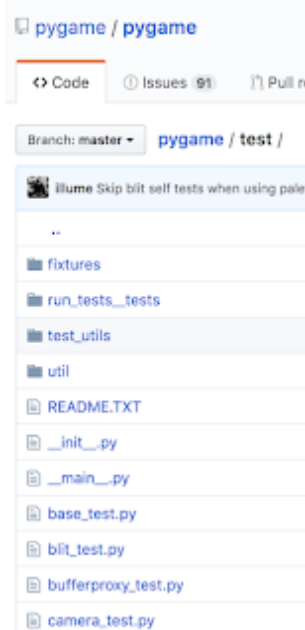
So you've forked the repo, and now you can clone your own copy of the git repo locally.

```
$ git clone https://github.com/YOUR-USERNAME/pygame
$ cd pygame/
$ python test/draw_test.py
...
-----
Ran 3 tests in 0.007s

OK
```

You'll see all of the tests in the test/ folder.

Browse the test folder online: <https://github.com/pygame/pygame/tree/main/test>



If you have an older version of pygame, you can use this little program to see the issue.

There is some more extensive documentation in the test/README file. Including on how to write a test that requires manual interaction.

19.4 Standard unittest module.

pygame uses the standard python unittest module. With a few enhancements to make it nicer for developing C code.

Fun fact: pygame included the unit testing module before python did.

We will go over the basics in this guide, but for more detailed information please see:

<https://docs.python.org/3/library/unittest.html>

19.5 How to run a single test?

Running all the tests at once can take a while. What if you just want to run a single test?

If we look inside draw_test.py, each test is a class name, and a function. There is a “DrawModuleTest” class, and there should be a “def test_ellipse” function.

19.5.1 So, let's run the test...

```
~/pygame/ $ python test/draw_test.py DrawModuleTest.test_ellipse
Traceback (most recent call last):
...
AttributeError: type object 'DrawModuleTest' has no attribute 'test_ellipse'
```

Good. This fails. It's because we don't have a test called "def test_ellipse" in there yet. What there is, is a method called 'todo_test_ellipse'. This is an extension pygame testing framework has so we can easily see which functionality we still need to write tests for.

```
~/pygame/ $ python -m pygame.tests --incomplete
```

...

FAILED (errors=39)

Looks like there are currently 39 functions or methods without a test. Easy pickings.

19.6 Python 3 to the rescue.

Tip: Python 3.7 makes it easier to run tests with the magic "-k" argument. With this you can run tests that match a substring. So to run all the tests with "ellipse" in their name you can do this:

```
~/pygame/ $ python3 test/draw_test.py -k ellipse
```

19.7 Digression: Good first issue, low hanging fruit, and help wanted.

Something that's easy to do.

A little digression for a moment... what is a good first issue?

Low hanging fruit is easy to get off the tree. You don't need a ladder, or robot arms with a claw on the end. So I guess that's what people are talking about in the programming world when they say "low hanging fruit".

Many projects keep a list of "good first issue", "low hanging fruit", or "help wanted" labeled issues. Like the [pygame "good first issue"](#) list. Ones other people don't think will be all that super hard to do. If you can't find any on there labeled like this, then ask them. Perhaps they'll know of something easy to do, but haven't had the time to mark one yet.

One little trick is that writing a simple test is quite easy for most projects. So if they don't have any marked "low hanging fruit", or "good first issue" go take a look in their test folder and see if you can add something in there.

Don't be afraid to ask questions. If you look at an issue, and you can't figure it out, or get stuck on something, ask a nice question in there for help.

19.7.1 Digression: Contribution guide.

There's usually also a contribution guide. Like the [pygame Contribute](#) wiki page. Or it may be called developer docs, or there may be a CONTRIBUTING.md file in the source code repository. Often there is a separate place the developers talk on. For pygame it is the pygame mailing list, but there is also a chat server which is a bit more informal.

19.8 A full example of a test.

The unittest module arranges tests inside functions that start with “**test_**” that live in a class. Here is a full example:

```
import unittest

class TestEllipse(unittest.TestCase):

    def test_ellipse(self):
        import pygame.draw
        surf = pygame.Surface((320, 200))
        pygame.draw.ellipse(surf, (255, 0, 0), (10, 10, 25, 20))

if __name__ == '__main__':
    unittest.main()
```

You can save that in a file yourself(test_draw1.py for example) and run it to see if it passes.

19.9 Committing your test, and making a Pull Request.

Here you need to make sure you have “git” setup. Also you should have “forked” the repo you want to make changes on, and done a ‘git clone’ of it.

```
# create a "branch"
git checkout -b my-draw-test-branch

# save your changes locally.
git commit test/draw_test.py -m "test for the draw.ellipse function"

# push your changes
git push origin my-draw-test-branch
```

Here we see a screenshot of a terminal running these commands.

When you push your changes, it will print out some progress, and then give you a URL at which you can create a “pull request”.

When you git push it prints out these instructions:

```
remote: Create a pull request for 'my-draw-test-branch' on GitHub by visiting:
remote:      https://github.com/YOURUSERNAME/pygame/pull/new/my-draw-test-branch
```

You can also go to your online fork to create a pull request there.

19.10 Writing your pull request text.

When you create a pull request, you are saying “hey, I made these changes. Do you want them? What do you think? Do you want me to change anything? Is this ok?”

It’s usually good to link your pull request to an “issue”. Maybe you’re starting to fix an existing problem with the code.

19.10.1 Testing the result with assertEquals.

How about it we want to test if the draw function actually draws something?

Put this code into test_draw2.py

```
import unittest

class TestEllipse(unittest.TestCase):

    def test_ellipse(self):
        import pygame.draw
        black = pygame.Color('black')
        red = pygame.Color('red')

        surf = pygame.Surface((320, 200))
        surf.fill(black)

        # The area the ellipse is contained in, is held by rect.
        #
        # 10 pixels from the left,
        # 11 pixels from the top.
        # 225 pixels wide.
        # 95 pixels high.
```

(continues on next page)

(continued from previous page)

```

rect = (10, 11, 225, 95)
pygame.draw.ellipse(surf, red, rect)

# To see what is drawn you can save the image.
# pygame.image.save(surf, "test_draw2_image.png")

# The ellipse should not draw over the black in the top left spot.
self.assertEqual(surf.get_at((0, 0)), black)

# It should be red in the middle of the ellipse.
middle_of_ellipse = (125, 55)
self.assertEqual(surf.get_at(middle_of_ellipse), red)

if __name__ == '__main__':
    unittest.main()

```

19.11 What is a git for? Jargon.

jargon - internet slang used by programmers. Rather than use a paragraph to explain something, people made up all sorts of strange words and phrases.

git - for sharing versions of source code. It lets people work together, and provides tools for people to.

pull request (PR) - “Dear everyone, I request that you **git pull** my commits.”. A pull request is a conversation starter. “Hey, I made a PR. Can you have a look?”. When you “**git push**” your commits (upload your changes).

unit test - **does this thing(unit) even work(test)!!!?** A program to test if another program works (how you think it should). Rather than test manually over and over again, a unit test can be written and then automatically test your code. A unit test is a nice example of how to use what you’ve made too. So when you do a pull request the people looking at it know what the code is supposed to do, and that the machine has already checked the code works for them.

assert - “assert 1 == 1”. An assert is saying something is true. “I assert that one equals one!”. You can also assert variables.

Contents

- *The arduino code*
- *python pygame code*

Taking a byte of bits of Serial, along with Green screen with Sam, pygame, and an Arduino hooked up to a light sensor and a motor thing.

Coffee too? Naturally.

Where do we begin? At the end of course. A video of the result...

THE ARDUINO CODE

```
#include <Servo.h>
Servo myservo;
int potpin = 0;
int val;

void setup() {
  Serial.begin(9600);
  myservo.attach(9);
}

void loop() {
  val = analogRead(potpin);
  Serial.println(val);
  val = map(val, 62, 540, 0, 179);
  myservo.write(val);
  delay(15);
}
```


PYTHON PYGAME CODE

Extended the “green eggs and ham” code I made recently to read serial data from the Arduino, and paint the screen different shades of green, whilst also making some silly sounds depending on the value read from the light sensor.

```
import os
import pygame
import serial

# guessing serial ports names. Linux or macosx.
# '/dev/tty.usbserial'
ser = serial.Serial('/dev/tty.usbmodem1411',
                    9600,
                    timeout = 0)

serial_buffer = ""

# an event number for the SERIAL
SERIAL = pygame.USEREVENT + 1

from pygame.locals import *
Event = pygame.Event
split = os.path.split

pygame.mixer.pre_init(44100, 8, 2, 1024)
pygame.init()
screen = pygame.display.set_mode((640, 480))
message = "Press q to quit, b for blue, r for red."
pygame.display.set_caption(message)

import pygame.examples
example_path = split(pygame.examples.__file__)[0]
example_data = os.path.join(example_path, "data")
sounds = [pygame.mixer.Sound("wiff.wav"),
          pygame.mixer.Sound("punch.wav"),
          ]

going = True
while going:
    serial_data = ser.read()
    #print (repr(serial_data))
    while serial_data:
        serial_buffer += serial_data
        # if there is a new line in it, then
        if "\r\n" in serial_buffer:
            #print (repr(serial_buffer))
```

(continues on next page)

(continued from previous page)

```
        evt = Event(SERIAL,
                    {'line': serial_buffer})
        pygame.event.post(evt)
        serial_buffer = ""
    serial_data = ser.read()

events = pygame.event.get()

for event in events:
    print (event)
    if event.type == KEYDOWN and event.key == K_q:
        going = False
    if event.type == KEYDOWN and event.key == K_b:
        screen.fill(Color("blue"))
    if event.type == KEYDOWN and event.key == K_r:
        screen.fill(Color("red"))
    if event.type == SERIAL:
        # read the line from the serial event.
        print (repr(event.line))

        # clean up, and do sanity checking.
        # It could be corrupt or garbage.
        line = event.line.replace("\r\n", "")
        line = line.replace("\n", "")
        line = line.replace("\r", "")
        # it could be empty.
        if line:
            val = int(line)
            if val < 1024 and val > 0:
                print (repr(val))
                # map 0 and 1023 of analog read to
                # 0-255 colour colour range.
                green = int((val/float(1023))*255 )
                screen.fill((0, green, 0))
                if val < 100:
                    sounds[0].play()
                if val > 300:
                    sounds[1].play()

pygame.display.flip()

ser.close()
```

LET'S MAKE A SHIT JAVASCRIPT INTERPRETER!

Part one.

Part two.

Part three.

Contents

- *Let's make a shit javascript interpreter! Part one.*
 - *Tokenising*
 - * *We can has vegetarian cheeseburger... but how can we parse javascript?*
 - * *Goat driven development*
 - * *So where to begin?*
 - * *A Token data structure.*
 - * *Writing the tokeniser*
 - * *Our homework*
 - * *Until next time...*



LET'S MAKE A SHIT JAVASCRIPT INTERPRETER! PART ONE.

As a learning exercise, I've begun writing a [javascript ECMAScript](#) interpreter in [python](#). It doesn't even really exist yet, and when it does it will run really slowly, and not support all js features.

So... let's make a "from scratch", all parsing, all dancing, shit interpreter of our very own!

Teaching something is a great way to learn. Also writing things on my blog always gets good 'comments', hints, tips, plenty of heart, and outright HATE from people. All useful and entertaining :)

23.1 Tokenising

So to start with, we need something to turn the .js files into a list of tokens. This type of program is called a tokeniser.

From some *javascript* like this:

```
function i_can_has_cheezbrgr () {return 'yum';};
```

Into a *Token list* something like this:

```
[
  {"type": "name",
   "value": "function",
   "from": 0,
   "to": 8},
  {"type": "name",
   "value": "i_can_has_cheezbrgr",
   "from": 9,
   "to": 28},
  {"type": "operator",
   "value": "(",
   "from": 29,
   "to": 30},
  {"type": "operator",
   "value": ")",
   "from": 30,
   "to": 31},
  {"type": "operator",
   "value": "{",
   "from": 32,
   "to": 33},
  {"type": "name",
   "value": "return",
   "from": 33,
   "to": 39},
```

(continues on next page)

(continued from previous page)

```
{ "type": "string",  
  "value": "yum",  
  "from": 40,  
  "to": 45},  
{ "type": "operator",  
  "value": ";",  
  "from": 45,  
  "to": 46},  
{ "type": "operator",  
  "value": ")",  
  "from": 46,  
  "to": 47},  
{ "type": "operator",  
  "value": ";",  
  "from": 47,  
  "to": 48}  
]
```

Wikipedia has a page on [Parsing](#) (also see [List_of_unusual_articles](#) for some other background information).

“*Tokenization* is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.” – wikipedia [Lexical_analysis#Token](#) page.

23.1.1 We can has vegetarian cheeseburger. . . but how can we parse javascript?

To the rescue, comes uncle Crockford the javascript guru of [jslint](#) fame. He wrote this lovely article: <http://javascript.crockford.com/tdop/tdop.html>. The ideas come from a 1973 paper called “[Top Down Operator Precedence](#)”. The Crockford article is great, since it is free, short, and well written javascript. Unlike the 1973 paper it gets the ideas from. . . which is behind a paywall, long, and uses a 1973 language called “(L(i,(s,(p))))”. As well as being short and simple. . . [Phil Hassey](#) used “Top Down Operator Precedence” and this article on his journey [making tinypy](#).

23.1.2 Goat driven development



Just as Phil did with tinypy, I’m going to use **Goat Driven Development**. Well, I’m not even sure what Goat Driven Development is. . . so maybe not.

Another python using dude, Fredrik Lundh, wrote some articles on “[Simple Top-Down Parsing in Python](#)” and [Top-Down Operator Precedence Parsing](#).

Also see [Eli Bendersky’s](#) article on Top Down Operator Precedence.

23.1.3 So where to begin?

After reading those articles a few times... scratching my head 13 times, making 27 hums, a few haaarrrrs, one hrmmmm, and four lalalas...

light bulb: A brilliant plan!

Eli Bendersky implements a full tokeniser, and parser for simple expressions like “1 + 2 * 4”.

Let’s copy this approach, but simplify it even more. Our first step is to make a tokeniser for a such an expression. That should be easy right?

23.1.4 A Token data structure.

Uncle Doug Crockford uses this structure for a token.

```
// Produce an array of simple token objects from a string.
// A simple token object contains these members:
//     type: 'name', 'string', 'number', 'operator'
//     value: string or number value of the token
//     from: index of first character of the token
//     to: index of the last character + 1
```

Here’s an example token from above:

```
{ "type": "name",
  "value": "i_can_has_cheezbrgr",
  "from": 9,
  "to": 28 }
```

23.1.5 Writing the tokeniser

Often a tokeniser is generated... or written by hand.

Fredrik Lundh writes a simple tokeniser using a regular expression.

```
>>> import re
>>> program = "1 + 2"
>>> [(number, operator) for number, operator in
...   re.compile("\s*(?:\d+|(.))").findall(program)]
[('1', '+'), ('', '+'), ('2', '+')]
```

This is a valid approach... but regexen blow up minds. Instead I’m going to write one using a state machine, in a big while loop with lots of ifs and elses.

23.1.6 Our homework

Write a tokeniser for simple expressions like “1 + 2 * 4”. Output a list of tokens like the javascript one does... eg.

```
{ "type": "name",  
  "value": "i_can_has_cheezbrgr",  
  "from": 9,  
  "to": 28 }
```

23.1.7 Until next time...

Really, I have no idea what I'm doing... but that's never stopped me before! It's going to be a shit javascript, but it will be **our shit javascript**.

Contents

- *Let's make a shit javascript interpreter! Part two.*
 - *Homework from part One - A simple tokeniser.*
 - *Our simple tokeniser*
 - *Code for shitjs.*
 - *What next? Parsing with the tokens of our simple expression.*
 - *What's new is old is new.*
 - *Manually stepping through the algorithm.*
 - *Exercises for next time*
 - *Until next time... Further reading (for the train, or the bathtub).*



LET’S MAKE A SHIT JAVASCRIPT INTERPRETER! PART TWO.

As a learning exercise, I’ve begun writing a JavaScript ECMAScript interpreter in python. It doesn’t even really exist yet, and when it does it will run really slowly, and not support all js features.

24.1 Homework from part One - A simple tokeniser.

We ended “Let’s make a shit JavaScript interpreter! Part One.” by setting some homework to create a tokeniser for simple expressions like “1 + 2 * 4”. Two readers sent in their versions of the tokenisers (ps, put a link to your home work results from Part One in the comments, and I’ll link to it here).

- <http://pastebin.com/P0sXjr65>
- <http://gist.github.com/460197>

24.2 Our simple tokeniser

```
operators = ['/', '+', '*', '-']
class ParseError(Exception):
    pass
def is_operator(s):
    return s in operators
def is_number(s):
    return s.isdigit()

def tokenise(expression):
    pos = 0
    for s in expression.split():
        t = {}
        if is_operator(s):
            t['type'] = 'operator'
            t['value'] = s
        elif is_number(s):
            t['type'] = 'number'
            t['value'] = float(s)
        else:
            raise ParseError(s, pos)

        t.update({'from': pos, 'to': pos + len(s)})
        pos += len(s) + 1
    yield t
```

```
>>> pprint(list(tokenise("1 + 2 * 4")))
[{'from': 0, 'to': 1, 'type': 'number', 'value': 1.0},
 {'from': 2, 'to': 3, 'type': 'operator', 'value': '+'},
 {'from': 4, 'to': 5, 'type': 'number', 'value': 2.0},
 {'from': 6, 'to': 7, 'type': 'operator', 'value': '*'},
 {'from': 8, 'to': 9, 'type': 'number', 'value': 4.0}]
```

24.3 Code for shitjs.

You can follow along with the code for shit js at:

- <http://pypi.python.org/pypi/shitjs>
- `pip install shitjs`
- <https://bitbucket.org/illume/shitjs>
- `hg clone https://illume@bitbucket.org/illume/shitjs`

After you have installed it, `shitjs.part1` is the package for the part1 homework.

24.4 What next? Parsing with the tokens of our simple expression.

Since we have some tokens from the input, we can now move onto the parser. Remember that we are not making a parser for all of javascript to start with, we are starting on a simple expressions like “1 + 2 * 4”. As mentioned in Part One, we are using an algorithm called “Top Down Operator precedence”. Where actions are associated with tokens, and an [order of operations](#). Here you can see the precedence rule (order of operations) with parenthesis around the (1 + 2) addition changes the result.

```
>>> 1 + 2 * 4
9
>>> (1 + 2) * 4
12
```

A number is supplied for the left, and the right of each token. These numbers are used to figure out which order the operators are applied to each other. So we take our token structure from `tokenise()` above, and we create some Token objects from them, and depending on their binding powers evaluate them.

24.5 What’s new is old is new.

The “Top Down Operator precedence” paper is from the 70’s. In the 70’s lisp programmers loved to use three letter variable names, and therefore the algorithm and the variable names are three letter ones. They also wore flares in the 70’s (which are back in this season) and I’m not wearing them, and I’m not using three letter variable names!

Sorry, I digress... So we call 'nud' prefix, and 'led' infix. We also call rbp right_binding_power, and lbp left_binding_power.

nud - prefix

led - infix

rbp - right_binding_power

lbp - left_binding_power

Prefix is to the left, and infix is to the right.

24.6 Manually stepping through the algorithm.

Let's manually step through the algorithm for the simple expression "1 + 2 * 4".

```
>>> pprint(list(tokenise("1 + 2 * 4")))
[{'from': 0, 'to': 1, 'type': 'number', 'value': 1.0},
 {'from': 2, 'to': 3, 'type': 'operator', 'value': '+'},
 {'from': 4, 'to': 5, 'type': 'number', 'value': 2.0},
 {'from': 6, 'to': 7, 'type': 'operator', 'value': '*'},
 {'from': 8, 'to': 9, 'type': 'number', 'value': 4.0}]
```

Let's give left binding powers to each of the token types.

- number - 0
- - operator - 10
- * operator - 20

Ok, so first we have a number token, with the value of 1.0. This is because in our shitjs so far all numbers are floats. Here is a log obtained by stepping through the expression.

```
('token', Literal({'to': 1, 'type': 'number', 'value': 1.0, 'from': 0}))
('expression right_binding_power: ', 0)
  ('token', OperatorAdd({'to': 3, 'type': 'operator', 'value': '+', 'from': 2}))
  ('left from prefix of first token', 1.0)
  ('token', Literal({'to': 5, 'type': 'number', 'value': 2.0, 'from': 4}))
  ('expression right_binding_power: ', 10)
    ('token', OperatorMul({'to': 7, 'type': 'operator', 'value': '*', 'from': 6}
    ↪))
    ('left from prefix of first token', 2.0)
    ('token', Literal({'to': 9, 'type': 'number', 'value': 4.0, 'from': 8}))
    ('expression right_binding_power: ', 20)
      ('token', End({}))
      ('left from prefix of first token', 4.0)
      ('leaving expression with left:', 4.0)
      ('left from previous_token.infix(left)', 8.0)
      right_binding_power:10: token.left_binding_power:0:
      ('leaving expression with left:', 8.0)
      ('left from previous_token.infix(left)', 9.0)
      right_binding_power:0: token.left_binding_power:0:
      ('leaving expression with left:', 9.0)
```

You can see that it is a recursive algorithm. Each indentation is where it is entering a new expression.

Also, see how it manages to use the binding powers to make sure that the multiplication of 2 and 4 is done first before the addition. Otherwise the answer might be $(1 + 2) * 4 == 12$! Not the correct answer 9 that it gives at the end.

The operations are ordered this way because the + operator has a lower left binding power than the * operator.

You should also be able to see from that trace that a tokens infix and prefix operators are used. The OperatorAdd for example, takes what is on the left and adds it to the expression of what is on the right with it's prefix operator. Here is an example Operator with prefix(left) and infix(right) methods.

```
class OperatorAdd(Token):
    left_binding_power = 10
    def prefix(self):
        return self.context.expression(100)
    def infix(self, left):
        return left + self.context.expression(self.left_binding_power)
```

Pretty simple right? You can see the infix method takes the value in from the left, and adds it to the expression of what comes on the right.

24.7 Exercises for next time

Make this work:

```
>>> evaluate("1 + 2 * 4")
9.0
```

(ps... if you want to cheat the code repository has my part 2 solution in it if you want to see. The code is very short).

24.8 Until next time... Further reading (for the train, or the bathtub).



Below is some further reading about parsers for JavaScript, and parsers in python.

- [simpleparse](#) and a json parser using simpleparse.
- [codetalker](#)
- [comparing python parser generators](#)
- [Extended Backus–Naur Form \(EBNF\)](#)

After following some of those links you may realise that we could probably make this shitjs interpreter in an easier way by reusing libraries. However if we wanted to do that, we'd just use an existing JavaScript implementation! Also our JavaScript wouldn't be shit, or from scratch.

Contents

- *Let's explore existing JavaScript implementations.*
 - *narcissus - js in js.*
 - *Spider monkey.*
 - *Google V8.*
 - *JSLint.*
 - *Rhino*
 - *KJS*
 - *JavaScriptCore, Squirrelfish, Nitro*
 - *Closed source JavaScript implementations*
 - *pypy javascript*
 - *So what have we learned then?*
 - *Exercise for next time*
 - *Further reading.*



Let's put the Research into R&D. I guess it should be called Shit Research to go along with the name of this [series \(p1\)](#) of [articles \(p2\)](#) I started one year ago. It takes a lot of time to read over hundreds of thousands of lines of undocumented C++ and java code, so part three took much longer than expected.

LET'S EXPLORE EXISTING JAVASCRIPT IMPLEMENTATIONS.

In this part we are going to take a small digression to look over other JavaScript implementations. I'll provide a short description of how each JavaScript implementation is made. We can use the Architectural knowledge from each implementation to inspire our own implementation.

Another benefit of researching each implementation is that we can find all the different tools they use. For example different test suites.

Make sure to read the URL's for each implementation to find out more information about each one.

There is a list of [ECMAScript implementations at wikipedia](#). We will not cover all of the ones listed there. If like me, you may spend a few hours or even days reading through the links from there.

25.1 narcissus - js in js.

Narcissus is a javascript implementation written in javascript (with some SpiderMonkey language extensions). It is written by the same author as the original JavaScript implementation back in 2005-2007 ([Brendan Eich](#)). In mid 2010 the Narcissus code has been taken up again by the Mozilla project to make researching JavaScript changes easier.

It is a good implementation to study, since it is fairly small, and is meant to be easy enough to read.

It uses a hand written parser, not a generated one.

[The original narcissus source repository](#).

The new [repository of narcissus](#), and [two articles](#) about it.

There is a port to python of the narcissus parser called [pynarcissus](#). The authors of pynarcissus found it difficult to port the rest of the interpreter since it relies on JavaScript features itself.

It weighs in at about 7000 lines of code counted with `wc -l`. I had to count lines in this way since my SLOC counter does not seem to count javascript.

25.2 Spider monkey.

[Spider Monkey](#) is the original JavaScript implementation used by Netscape, and the Mozilla project.

SpiderMonkey is a production grade, high quality JavaScript implementation.

It also has excellent documentation. Especially the [js/src/README.html](#) file which includes a design walk through.

“The compiler consists of a recursive-descent parser and a random-logic rather than table-driven lexical scanner. Semantic and lexical feedback are used to disambiguate hard cases such as missing semicolons, assignable expressions (“lvalues” in C parlance), etc. The parser generates bytecode as it parses, using fixup lists for downward branches and code buffering and rewriting for exceptional cases such as for loops. It attempts no error recovery. The interpreter executes the bytecode of top-level scripts, and calls itself indirectly to interpret function bodies (which are also scripts). All state associated with an interpreter instance is passed through formal parameters to the interpreter entry point; most implicit state is collected in a type named JSContext. Therefore, all API and almost all other functions in JSRef take a JSContext pointer as their first argument. The

decompiler translates postfix bytecode into infix source by consulting a separate byte-sized code, called source notes, to disambiguate bytecodes that result from more than one grammatical production.”””

25.3 Google V8.

Written in C++, javascript and assembler. Uses hidden classes, and generates machine code at run time.

The parser is hand written parser in C++. It's not generated.

'Prepares', which creates tokens. Then creates an AST by parsing. Finally compiling the AST. parser.css is where all the parsing happens.

The projects documentation is quite limited - so reading the source is the best way to get in there. There are some videos which describe the architecture, and engineering decisions behind the choices taken.

<http://code.google.com/p/v8/>

V8 weighs in at a mighty 605,962 SLOC. Making it the largest, biggest, badest V8 JavaScript engine around.

25.4 JSLint.

JSLint is written in JavaScript and uses a TDOP approach to the parser like we are using. I won't discuss this, since it is documented well elsewhere.

25.5 Rhino

Rhino can run as either an interpreter or a java byte code generator. It's hosted by the mozilla project (who make firefox). It complements their C++ implementation (spidermonkey) and their JavaScript implementation (narcissus).

There are a few things which show it is a very mature as well as modern implementation. The project was started in 1999, and has been developed ever since. It has partial support for JavaScript 1.8, and ECMAScript 5 standards. Weighing in at 50,551 source lines of Java code (SLOC), it can be considered a very large code base. Another modern feature is that it supports the CommonJS javascript module standard. Despite still being hosted in cvs, it's still being maintained, and features are being added regularly.

The Rhino JavaScript team maintains a library of tests and other documentation for JavaScript. Tests are being shared with the other JavaScript implementations within the mozilla spidermonkey project. There have also been parts which have been ported from spidermonkey - such as the hand written parser.

A handwritten scanner they call TokenStream creates tokens, and then parses those into an AST. Despite being mostly hand written, some parts are generated. Specifically the stringToKeyword method, which detects keywords is generated somehow.

The documentation of the architecture of the project is limited. There is however some API documentation. With a couple of modifications to some ant build files I was able to build it, as well as even make a few small modifications.

The [wikipedia Rhino](#) page has some great information on the rhino javascript engine.

25.6 KJS

KJS is the KDE JavaScript implementation for the konqueror browser. It was the parent of the JavaScript implementations done by Apple Computer, inc. I won't go into any detail on this one, since I'll cover JavaScriptCore instead. KJS is written in C++, for the QT library.

25.7 JavaScriptCore, Squirrelfish, Nitro

You can browse the source here: <http://trac.webkit.org/browser/trunk/Source/JavaScriptCore>

This uses a hand written lexer(tokeniser), and a hand written parser. The code structure of the parser and lexer looked eerily familiar. The code base is mostly written in C++ and is quite massive. 140,837 SLOC

There is lots of platform specific code, but it also has a jit, uses byte code, and an interpreter. There is also lots of development code in there for things like debuggers, and profilers.

25.8 Closed source JavaScript implementations

There are a few JavaScript implementations that are closed source. The two main ones in widespread use are the ones from Microsoft, and the ones from Opera.

They have however published papers and blog posts about their implementations. I won't cover them any more, because not as much can be learned without the source code.

25.9 pypy javascript

The pypy project started a javascript interpreter now too.

<https://bitbucket.org/pypy/lang-js/src/de89ec32a7dc/js/javascript-interpreter.txt>

The description of the project mentions it's currently using the spider monkey parser, but it appears to generate one using a parser generator provided by pypy. Using a EBNF grammar file. It also creates an AST.

It works for some simple javascript programs that don't use the javascript standard library. I'm not sure of the future of the project, since it appears it was a GSOC project which has now finished, so there might not be any full time developers left on it.

It's written in RPython (a restricted subset of python) and python. Running on top of pypy, it should theoretically be able to take advantage of that platforms jit and garbage collector.

This project makes use of some JavaScript tests and benchmarks from other projects. Specifically some benchmarks from v8, and the language shootout website. It also includes the "ECMA 262 Edition 1" tests.

It weighs in at 5452 Source Lines of Code (SLOC). Which is much smaller, but the implementation is also not complete, so that is to be expected.

25.10 So what have we learned then?

We see that most of the implementations use a hand written parser. We also see that the implementations in js and python are much smaller. So despite them being incomplete, I think it proves that it should be feasible to make our shit interpreter in python. We don't need half a million lines of C++ to do our project.

We have also learned that there are test suites available, which should help us out a lot. In fact, many of the implementations share the test suites. Having a test suite already available makes it way easier to write an implementation of something yourself. It acts as a guide to development, and also reduces the time for testing since a lot of it can be automated.

25.11 Exercise for next time

Choose One(1) of the implementations, build it, run it, and modify it slightly to do something different. Try and run the tests that come with it.

25.12 Further reading.

This whole article is “further reading”, but we can never have too much to read. Can we!?

This time, instead of reading it on the train or in the bath tub - may I suggest reading these on a couch?

- [Functions and execution contexts in JavaScript](#)
- [jslex, a javascript lexer written in python](#)
- <http://ruslanspivak.com/2011/05/02/slimit-a-javascript-minifier-to-be-is-released/> (pypi slimit)
- [UglifyJS](#)
- [Closure Compiler](#)

PYTHON BEST PRACTICES

Every project is somewhat different. Every reason to code is different.

Therefore best practices should be different. You won't do things the same way hacking up a weekend game for fun as you would grinding away in a cubicle implementing an aircraft safety system.

There's a lot of information out there about python. Lots of tools to choose between.

This guide aims to point you in the right direction on a few topics.

26.1 Testing

Use pytest because

- No boilerplate
- Failure reports
- Scalability
- Fixtures
- Mocking/Monkeypatching
- Mature with great plugins

26.1.1 Integration tests

Separate integration (functional) tests from unit tests.

Unit tests should be quick to run, and integration tests should test the integration with external APIs or systems.

26.1.2 Code coverage

Use code coverage reporting to make sure code is at least tested. Track and enforce a minimum standard in CI.

26.2 Documentation

It's fine, and good to write just a README.md file.

It's more important that you have:

- what it is, what it does, who you are, and why?
- examples of how to use it
- how to learn it, and how to use it to do certain tasks
- detailed reference

Write documentation with [Sphinx](#) and RestructuredText.

You can write narrative style documentation, and also generate API documentation. If you want to do something more than a basic README.md in your repo, then this is the way to go.

26.2.1 Type documentation

First, remember that type hints are not needed for python and for all types of projects. But they can be useful.

Document types with the typing module, rather than specify them in docstrings. Because then the types can be checked with a tool called mypy in CI and in some code editors.

From sphinx-autodoc-typehints [sphinx-autodoc-typehints](#) ...

```
from typing import Union

def format_unit(value: Union[float, int], unit: str) -> str:
    """ Formats the given value as a human readable string
        using the given units.

        :param value: a numeric value
        :param unit: the unit for the value (kg, m, etc.)
    """
    return '{} {}'.format(value, unit)
```

There is an [example](#) of how to document your python functions

26.3 Literate programming

Experiments and live documentation can be done with [jupyter notebooks](#)

This is a great tool for experiments, notebooks, and live documentation, and visualizations.

26.4 Code quality and formatting

Use [pylint](#) for linting, and [black](#) for code formatting.

Especially for developers new to python, pylint gives a lot of good advice on how to improve your code.

It can catch many simple errors before they get checked into source control or go into production.

A code formatter like [black](#) is useful in large teams so as to avoid having to talk about code formatting.

26.5 Package your code

The [python packaging guide](#) explains how to package python code.

- share your code with others
- a file structure all tools expect

26.6 Manage dependencies

with [poetry](#). Because poetry is safe, easy, and addresses the whole workflow.

The [pip](#) tool is still good to learn. However it has a few dangerous edges if you are a newbie. It also does not address the whole workflow of using and publishing packages.

Work inside a virtual environment to keep projects separated. Because each project may require different dependencies installing them system wide will not work.

26.7 A private python index

This is only needed if you want to keep some of your code private.

Private Python packages can be stored in a private python index. [devpi](#)

Devpi is also useful because it can act as a local mirror and cache. This can make deploys quicker and more reliable.

You can also use github, or another private git code repository and use these for packages.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`