# Good Coding Practices

# [insert clever comment here]

# SDT 2018-07-30

# @authors: Laura Colbran and Mary Lauren Benton

Why though?

Easy for others and future-you to use later

Avoid misleading others or causing … unanticipated outcomes

Faster to add functionality in the future

Readability

You stare at scripts all day, might as well make them nice

So no one will add you to a "worst code I ever saw" reddit thread

And yes, those really do exist

(really, the better question is 'why not?')

Whitespace is your ~~friend~~ really good acquaintance.

```
import sys, os
# oh look a function now
def my_func(to_add, to_sub, to_add_also,
  man_this_is_getting_long,
     oh_this_too):
     answer = ( 42 + to_add -
     to_sub + to_add_also *
     man_this_is_getting_long)
     return answer + oh_this_too
x                        = 10 * 6 - 1 / 3 + 7
really_really_long_name = "Bob"
print( "{}'s number is {}, but mine is {}.".format( really_really_long_name, x,
my_func( 1, 2, 3, 4, 5) ) )
```

Whitespace is your ~~friend~~ really good acquaintance.

```
import sys, os
# oh look a function now
def my_func(to_add, to_sub, to_add_also,
  man_this_is_getting_long,
     oh_this_too):
     answer = ( 42 + to_add -
     to_sub + to_add_also *
     man_this_is_getting_long)
     return answer + oh_this_too
x                        = 10 * 6 - 1 / 3 + 7
really_really_long_name = "Bob"
print( "{}'s number is {}, but mine is {}.".format( really_really_long_name, x,
my_func( 1, 2, 3, 4, 5) ) )
```

Please don't do this.

Whitespace is your ~~friend~~ really good acquaintance.

```
import sys
import os

# oh look a function now
def my_func(to_add, to_sub, to_add_also,
            man_this_is_getting_long,
            oh_this_too):
      answer = (42
               + to_add
               - to_sub
               + to_add_also
               * man_this_is_getting_long
               + oh_this_too)
      return answer

x = 10*6 - 1/3 + 7
really_really_long_name = "Bob"

print("{}'s number is {}, but mine is {}.".format(really_really_long_name,
                                                   x,
                                                   my_func(1, 2, 3, 4, 5)))
```

Whitespace is your ~~friend~~ really good acquaintance.

```
import sys
import os

# oh look a function now
def my_func(to_add, to_sub, to_add_also,
            man_this_is_getting_long,
            oh_this_too):
     answer = (42
              + to_add
              - to_sub
              + to_add_also
              * man_this_is_getting_long
              + oh_this_too)
     return answer

x = 10*6 - 1/3 + 7
really_really_long_name = "Bob"

print("{}'s number is {}, but mine is {}.".format(really_really_long_name,
                                                  x,
                                                  my_func(1, 2, 3, 4, 5)))
```

Whitespace is your ~~friend~~ really good acquaintance.

```
import sys
import os

# oh look a function now
def my_func(to_add, to_sub, to_add_also,
            man_this_is_getting_long,
            oh_this_too):
     answer = (42
               + to_add
               - to_sub
               + to_add_also
               * man_this_is_getting_long
               + oh_this_too)
     return answer

x = 10*6 - 1/3 + 7
really_really_long_name = "Bob"

print("{}'s number is {}, but mine is {}.".format(really_really_long_name,
                                                  x,
                                                  my_func(1, 2, 3, 4, 5)))
```

This is still a useless and inefficient program… but it is much prettier.

Thanks, PEP8 style guide!

https://www.python.org/dev/peps/pep-0008/

Whitespace is your ~~friend~~ really good acquaintance.

Use whitespace to separate functions or logical code chunks

If the language doesn't care about indentation (i.e. R), *use it anyway*.

Try to keep lines under ~80 characters.

You can test for PEP8 compliance with `pycodestyle`.

I won't yell at you anymore. But pycodestyle will.

```
$ pycodestyle --statistics -qq Python-2.5/Lib
232     E201 whitespace after '['
599     E202 whitespace before ')'
631     E203 whitespace before ','
842     E211 whitespace before '('
2531    E221 multiple spaces before operator
4473    E301 expected 1 blank line, found 0
4006    E302 expected 2 blank lines, found 1
165     E303 too many blank lines (4)
325     E401 multiple imports on one line
3615    E501 line too long (82 characters)
612     W601 .has_key() is deprecated, use 'in'
1188    W602 deprecated form of raising exception
```

I won't yell at you anymore. But pycodestyle will.

```
$ pycodestyle --statistics -qq Python-2.5/Lib
232     E201 whitespace after '['
599     E202 whitespace before ')'
631     E203 whitespace before ','
842     E211 whitespace before '('
2531    E221 multiple spaces before operator
4473    E301 expected 1 blank line, found 0
4006    E302 expected 2 blank lines, found 1
165     E303 too many blank lines (4)
325     E401 multiple imports on one line
3615    E501 line too long (82 characters)
612     W601 .has_key() is deprecated, use 'in'
1188    W602 deprecated form of raising exception
```

(It also doesn't like trailing whitespace.)

meaningful_var_names = "are so important!"

We want our code to read as easily as pseudocode.

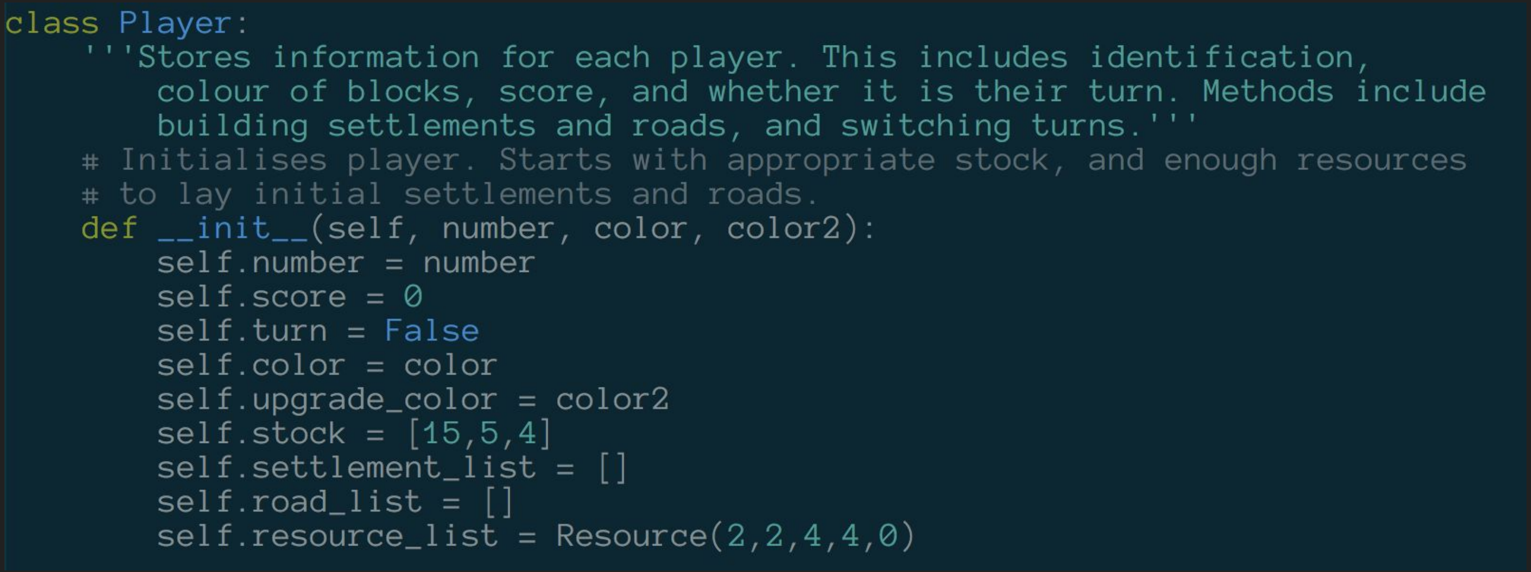
```
x = ['red', 'blue', 'yellow']
y = len(x)
print(y)
```

is more difficult to understand than:

```
colors = ['red', 'blue', 'yellow']
num_colors = len(colors)
print(num_colors)
```

meaningful_var_names = "are so important!"

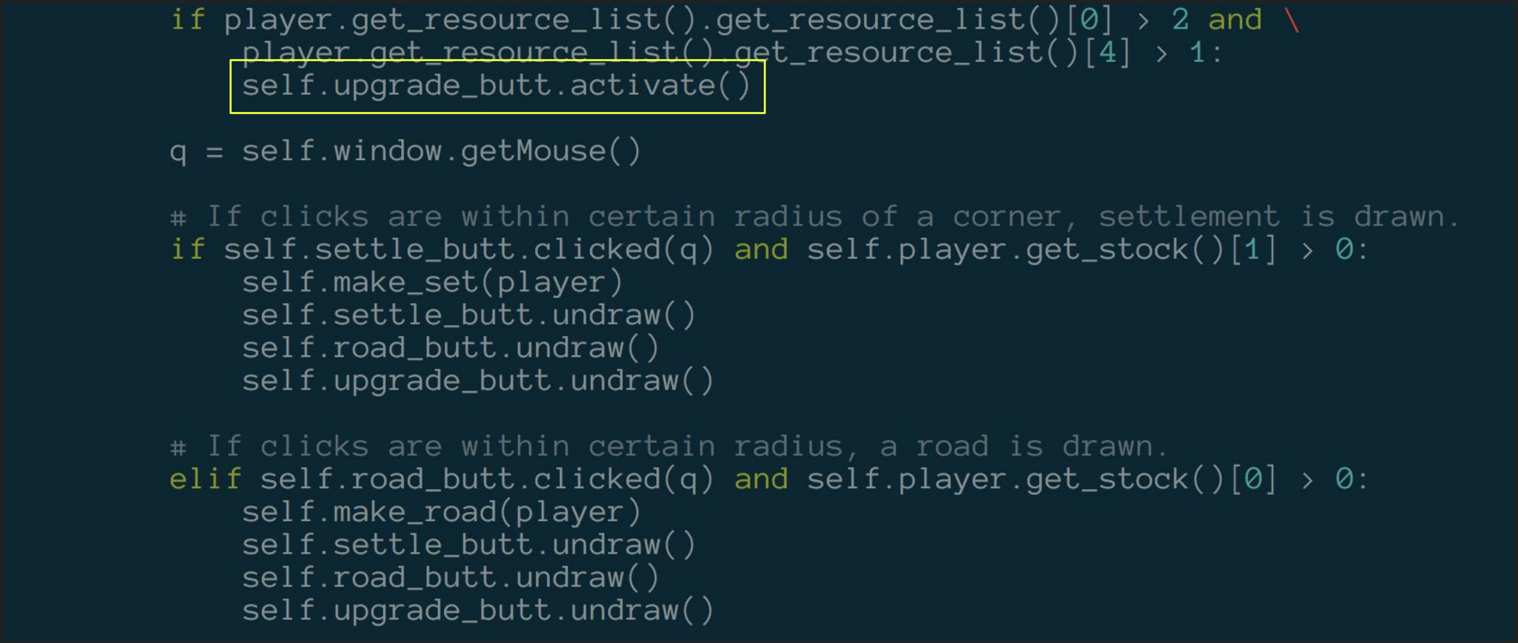
We want our code to read as easily as pseudocode.



```python
class Player:
    '''Stores information for each player. This includes identification,
        colour of blocks, score, and whether it is their turn. Methods include
        building settlements and roads, and switching turns.'''
    # Initialises player. Starts with appropriate stock, and enough resources
    # to lay initial settlements and roads.
    def __init__(self, number, color, color2):
        self.number = number
        self.score = 0
        self.turn = False
        self.color = color
        self.upgrade_color = color2
        self.stock = [15,5,4]
        self.settlement_list = []
        self.road_list = []
        self.resource_list = Resource(2,2,4,4,0)
```

meaningful_var_names = "can also be fun."

You can use abbreviations and still have a bit of fun (in moderation).



```python
if player.get_resource_list().get_resource_list()[0] > 2 and \
    player.get_resource_list().get_resource_list()[4] > 1:
    self.upgrade_butt.activate()

q = self.window.getMouse()

# If clicks are within certain radius of a corner, settlement is drawn.
if self.settle_butt.clicked(q) and self.player.get_stock()[1] > 0:
    self.make_set(player)
    self.settle_butt.undraw()
    self.road_butt.undraw()
    self.upgrade_butt.undraw()

# If clicks are within certain radius, a road is drawn.
elif self.road_butt.clicked(q) and self.player.get_stock()[0] > 0:
    self.make_road(player)
    self.settle_butt.undraw()
    self.road_butt.undraw()
    self.upgrade_butt.undraw()
```

```
def function():
    is_good = True
```

Why functions?

Efficiency
Debugging
Organization
Readability

def function():
is_good = True

Why functions?

Efficiency

Debugging

Organization

Readability

```
def main():
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday, dear Greg!")
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday, dear Greg!")
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday, dear Greg!")
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday to you!")
      print("Happy Birthday, dear Greg!")
      print("Happy Birthday to you!")
```

def function():
is_good = True

Why functions?

Efficiency

Debugging

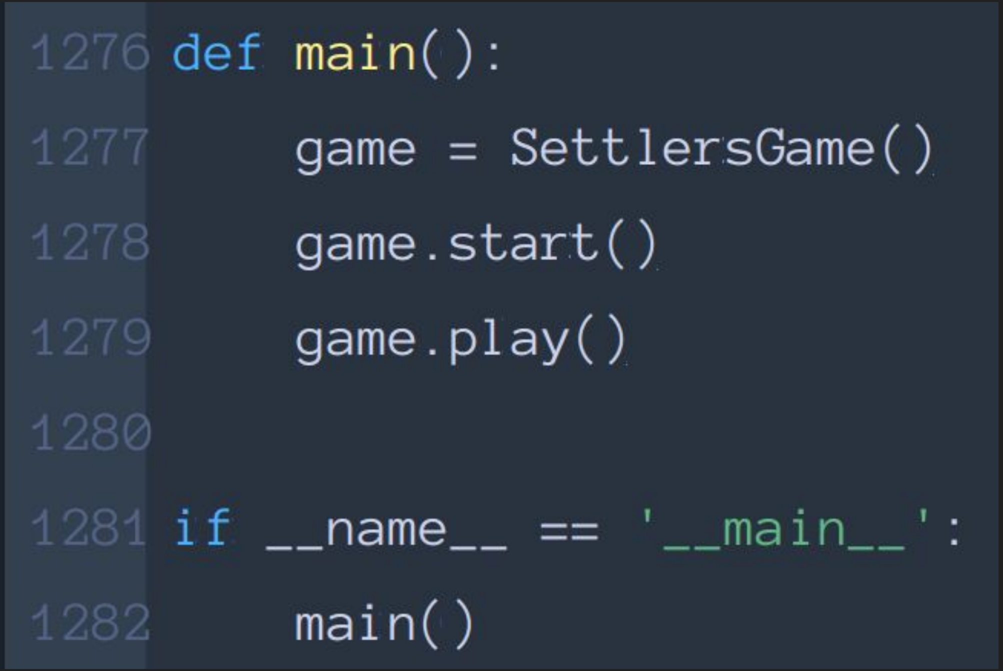
Organization

Readability

```
def happyBirthdayGreg():
     for line in range(0,4):
          if line == 2:
               print("Happy Birthday, dear Greg")
          else:
               print("Happy Birthday to you!")

def main():
     happyBirthdayGreg()
     happyBirthdayGreg()
     happyBirthdayGreg()
     happyBirthdayGreg()
```

```
def function():
    is_good = True
```

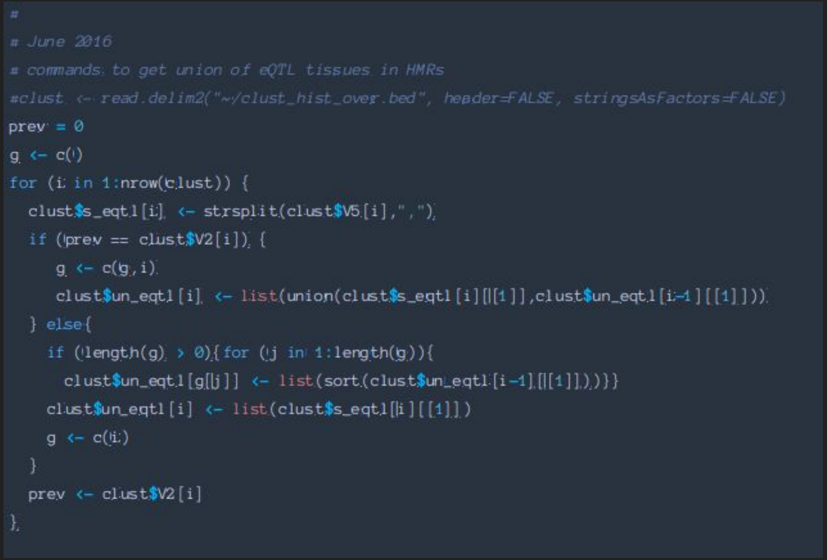
An ideal main() function looks like this:



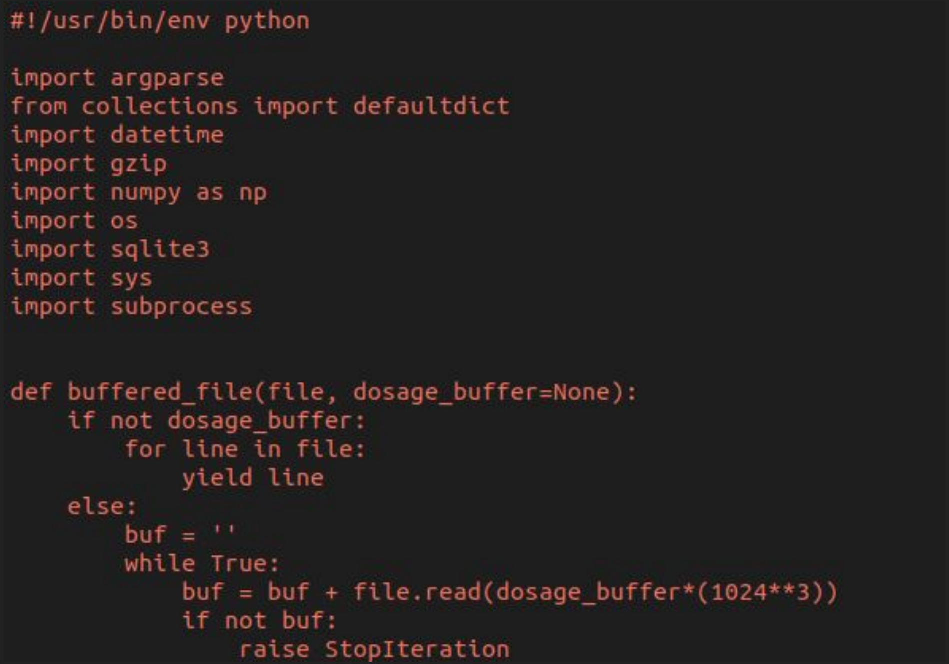
```python
def main():
    game = SettlersGame()
    game.start()
    game.play()

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

# comments are 100% necessary

But there is an art to doing it well...

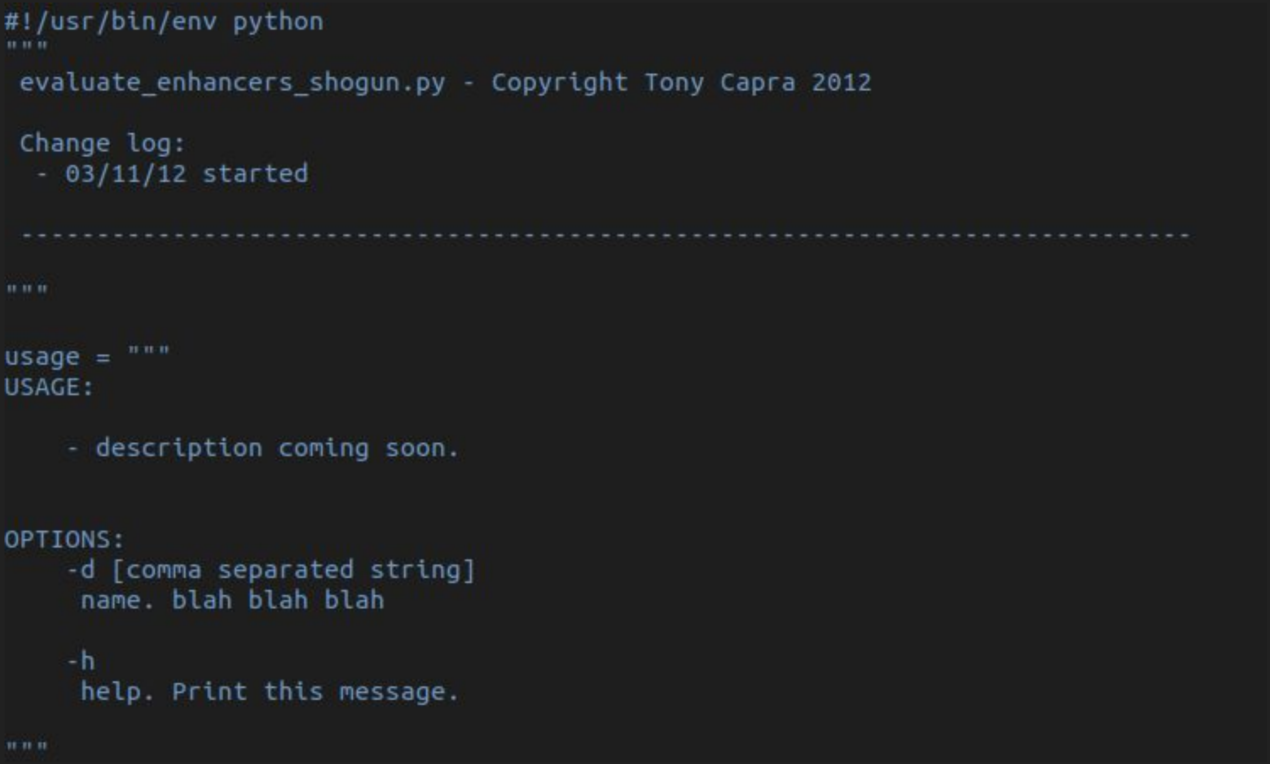


```r
#
# June 2016
# commands to get union of eQTL tissues in HMRs
#clust <- read.delim2("~/clust_hist_over.bed", header=FALSE, stringsAsFactors=FALSE)
prev = 0
g <- c()
for (i in 1:nrow(clust)) {
  clust$s_eqtl[i] <- strsplit(clust$V5[i],",")
  if (prev == clust$V2[i]) {
     g <- c(g,i)
     clust$un_eqtl[i] <- list(union(clust$s_eqtl[i][[1]],clust$un_eqtl[i-1][[1]]))
  } else{
    if (length(g) > 0){ for (j in 1:length(g)){
      clust$un_eqtl[g[j]] <- list(sort(clust$un_eqtl[i-1][[1]]))}}
    clust$un_eqtl[i] <- list(clust$s_eqtl[i][[1]])
    g <- c(i)
  }
  prev <- clust$V2[i]
}
```



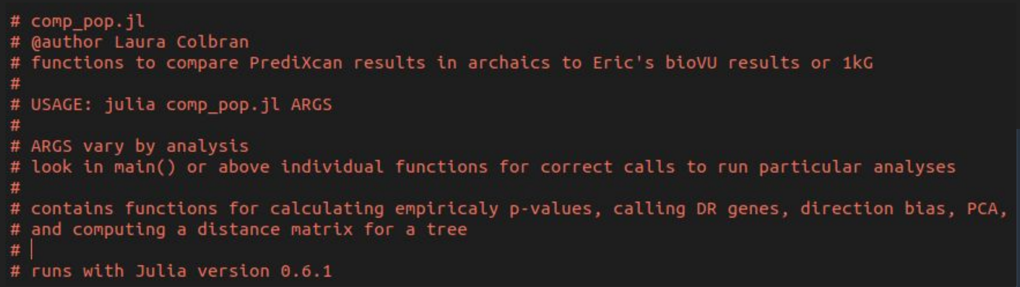
```python
#!/usr/bin/env python

import argparse
from collections import defaultdict
import datetime
import gzip
import numpy as np
import os
import sqlite3
import sys
import subprocess


def buffered_file(file, dosage_buffer=None):
    if not dosage_buffer:
        for line in file:
            yield line
    else:
        buf = ''
        while True:
            buf = buf + file.read(dosage_buffer*(1024**3))
            if not buf:
                raise StopIteration
```

# and just because a comment is present doesn't mean it's useful...



```
#!/usr/bin/env python
"""
 evaluate_enhancers_shogun.py - Copyright Tony Capra 2012

 Change log:
  - 03/11/12 started

 ----------------------------------------------------------------------------

"""

usage = """
USAGE:

    - description coming soon.


OPTIONS:
    -d [comma separated string]
     name. blah blah blah

    -h
     help. Print this message.

"""
```

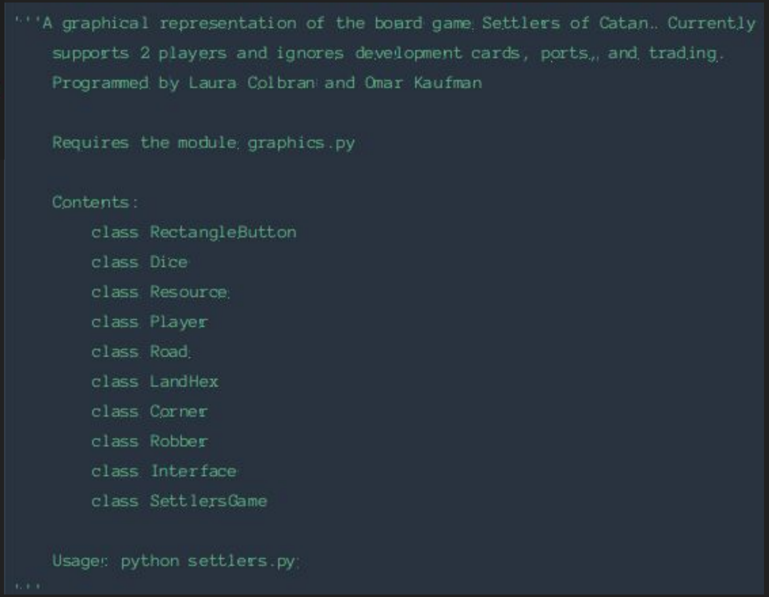
# so what should go in a comment?



```
# comp_pop.jl
# @author Laura Colbran
# functions to compare PrediXcan results in archaics to Eric's bioVU results or 1kG
#
# USAGE: julia comp_pop.jl ARGS
#
# ARGS vary by analysis
# look in main() or above individual functions for correct calls to run particular analyses
#
# contains functions for calculating empiricaly p-values, calling DR genes, direction bias, PCA,
# and computing a distance matrix for a tree
#
# runs with Julia version 0.6.1
```

'''

Header:
Author, description of contents, any required additional files, usage statement
'''



```
'''A graphical representation of the board game Settlers of Catan. Currently
    supports 2 players and ignores development cards, ports, and trading.
    Programmed by Laura Colbran and Omar Kaufman

    Requires the module graphics.py

    Contents:
        class RectangleButton
        class Dice
        class Resource
        class Player
        class Road
        class LandHex
        class Corner
        class Robber
        class Interface
        class SettlersGame

    Usage: python settlers.py
'''
```

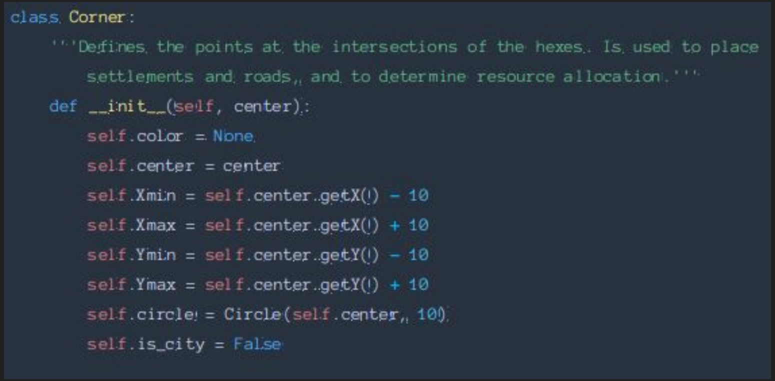
# so what should go in a comment?

'''

For each class and/or function:

What does it do? Does it do anything particularly weird or clever? What is the input it takes?

'''



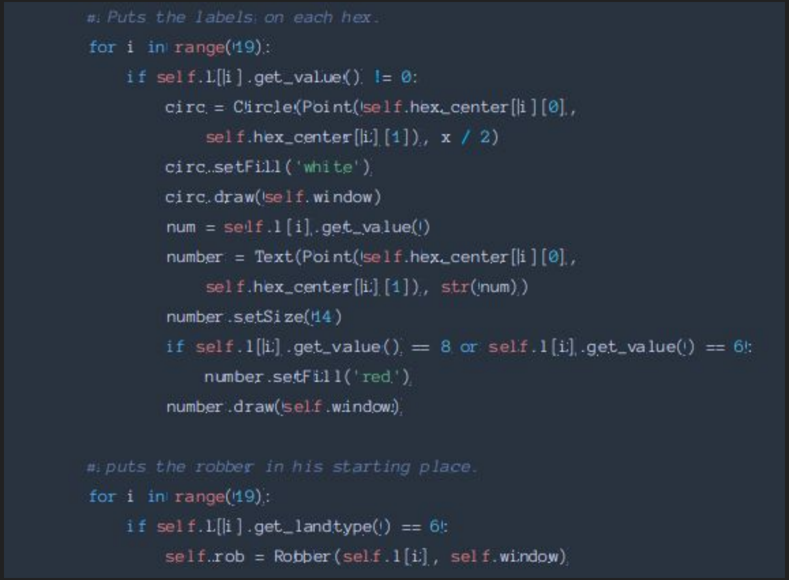
```python
class Corner:
    '''Defines the points at the intersections of the hexes. Is used to place
       settlements and roads, and to determine resource allocation.'''
    def __init__(self, center):
        self.color = None
        self.center = center
        self.Xmin = self.center.getX() - 10
        self.Xmax = self.center.getX() + 10
        self.Ymin = self.center.getY() - 10
        self.Ymax = self.center.getY() + 10
        self.circle = Circle(self.center, 10)
        self.is_city = False
```

# so what should go in a comment?

'''

If you’re feeling overachieving, or have a complicated algorithm, it is helpful to future debugging to put comments explaining each step of the code

'''



```
# Puts the labels on each hex.
for i in range(19):
    if self.l[i].get_value() != 0:
        circ = Circle(Point(self.hex_center[i][0],
            self.hex_center[i][1]), x / 2)
        circ.setFill('white')
        circ.draw(self.window)
        num = self.l[i].get_value()
        number = Text(Point(self.hex_center[i][0],
            self.hex_center[i][1]), str(num))
        number.setSize(14)
        if self.l[i].get_value() == 8 or self.l[i].get_value() == 6:
            number.setFill('red')
        number.draw(self.window)

# puts the robber in his starting place.
for i in range(19):
    if self.l[i].get_landtype() == 6:
        self.rob = Robber(self.l[i], self.window)
```

--arguments? Consider them parsed.

Using command line arguments will help avoid hard-coding values.

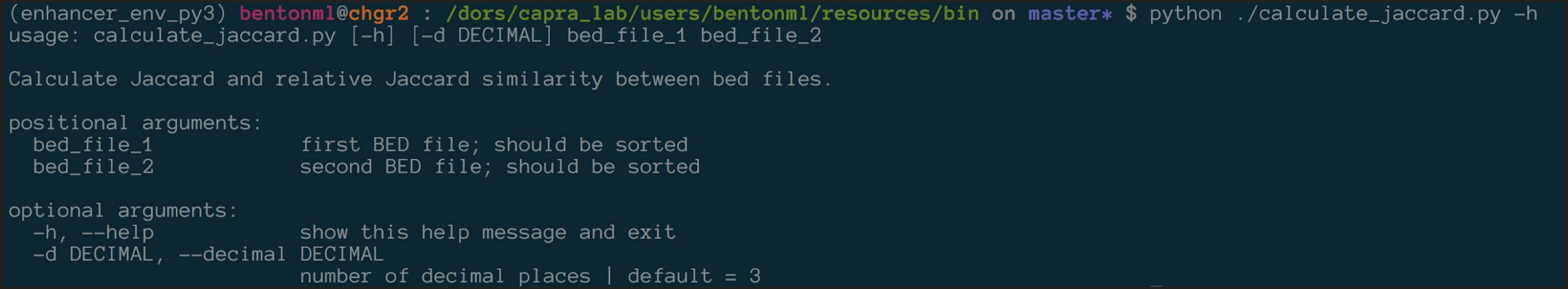
We'll use the `argparse` module in Python.

```python
import sys
import argparse
import pybedtools


###
#  arguments
###
arg_parser = argparse.ArgumentParser(description="Calculate Jaccard and relative Jaccard similarity between bed files.")

arg_parser.add_argument("bed_file_1", help='first BED file; should be sorted')
arg_parser.add_argument("bed_file_2", help='second BED file; should be sorted')
arg_parser.add_argument('-d', '--decimal', type=int, default=3, help='number of decimal places | default = 3')

args = arg_parser.parse_args()

# save parameters
A = args.bed_file_1
B = args.bed_file_2
DECIMAL = args.decimal
```

--arguments? Consider them parsed.

This has the added bonus of creating a usage/help message.

Access it with the `-h` option.



```
(enhancer_env_py3) bentonml@chgr2 : /dors/capra_lab/users/bentonml/resources/bin on master* $ python ./calculate_jaccard.py -h
usage: calculate_jaccard.py [-h] [-d DECIMAL] bed_file_1 bed_file_2

Calculate Jaccard and relative Jaccard similarity between bed files.

positional arguments:
  bed_file_1            first BED file; should be sorted
  bed_file_2            second BED file; should be sorted

optional arguments:
  -h, --help            show this help message and exit
  -d DECIMAL, --decimal DECIMAL
                        number of decimal places | default = 3
```

argparse essentials

```
# basic usage
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()  # do this after you add all arguments

# positional arguments
parser.add_argument("arg_name", help="description", type=str)
parser.arg_name  # will give you the value of arg_name

# optional arguments
parser.add_argument("-v", "--verbose", help="description", action="store_true")

parser.add_argument("-s", "--species", type=str, choices=['hg19', 'mm10'],
                    default='hg19', help="available species; default=hg19")
```

# In summary

```
def whitespace():
        print("use whitespace to separate functions or logical code chunks")
        print("use indentation, ALWAYS")
        print("keep lines under ~80 characters")

def variable_names():
        should_be_meaningful = True

def functions():
        return are_everything

def comments():
        # are essential

def input():
        if arguments > hard_coding:
                print("Right!")
```

The Zen of Python

```
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.

Although practicality beats purity.

. . .
```