Why Python?

Some of my favorite parts

- Clean, (relatively) straightforward syntax
- “Batteries included”
- Big community
- Versatile

Programs must be written for people to read, and only incidentally for machines to execute.

– Abelson & Sussman, Structure and Interpretation of Computer Programs

The reader shouldn't have to remember much

- Split code into functions that perform single, easily understood tasks
- Bonus: This makes them easier to test too

Names should be consistent and meaningful

Examples:

```
current_account_balance
comment_box_text
account.withdraw(20, ‘USD’)
```

Keep it DRY
(don't repeat yourself)

If you're copy-pasting code,

STOP

Document, but document the right things

Design and intent, not implementation

Bad:

```
# loop over movies in the list
for movie in list:
    # add genre to genre counts
    genres.update(movie.genre)
```

Document, but document the right things

Better

```
# Takes a list of Movie objects and returns a Counter of genre objects
def count_genres():
    ...
```

Docstrings even nicer

Docstrings are triple-quoted strings placed after a `def` or `class` that describe the functionality of that thing.

Can be used to automatically generate documentation

```
def count_genres():
    """Takes: a list of Movie objects
    returns: a Counter of genre objects
    """

    …
```

I Revisit *the zen* frequently.

```
import this
```

Beautiful is better than ugly

Looks gross? There may be a better way.

Explicit is better than implicit

Unpack values into meaningful names

```
today, tomorrow, today_plus_two = next_three_days(foo)
```

```
list_of_pairs = [("Mike","Sally"),("Larry","Gary")]
for index, (name_a, name_b) in enumerate(list_of_pairs):
    ...
```

Explicit is better than implicit

Use arguments well

Regular arguments are mandatory and have no default values.

Keyword arguments are optional and have default values.

```
def swap_random_pairs(data, n_swaps=1):
    ...
```

Explicit is better than implicit

Use arguments well

- If most of the time when you pass a value it is the exact same, consider moving it into a keyword argument for brevity.
- Don't anticipate which arguments you think you'll need-- write the function and add them as you need them

Readability Counts

Use “in” and "with" when possible

Good:

```
for thing in list:
    ...

if "cats" in sentence.split():
    ...

with open('file.txt') as file:
    ...
```

Bad:

```
for i in range(len(list)):
    ...

file = open('file.txt')
... # this code could raise an exception, leaving the file open
file.close()
```

Rely on truthiness

```
if numlist:
    ...
```

instead of

```
if len(numlist) > 0:
    ...
```

One statement per line

It's not a contest to make your code as compact as possible. Let it breathe.

BAD:

```
if foo == 'blah': do_something()
do_one(); do_two(); do_three()
```

Good:

```
if foo == 'blah':
    do_something()
do_one()
do_two()
do_three()
```

Errors should never pass silently

When things happen that are unexpected, have your code raise an exception.

```
if len(class_list) % 2 != 0:
    raise ValueError('List should have an even number of elements')
```

Never use bare `except:` statements

BAD:

```
try:
    str(x)
except:
    print "¯\_(ツ)_/¯"
```

Better:

```
try:
    str(x)
except TypeError:
    print "x could not be made a string."
    raise
```

Flat is better than nested

Make functions liberally

A quick bit about variables and "mutation" in Python

Variables in Python are references to things.
So if you assign a variable to another variable, it just makes both variables refer to the same thing:

```
>>> a = [1, 2]
>>> b = a
>>> a
[1, 2]
>>> b
[1, 2]
```

Most functions in Python take something, return a result, and don't change the thing you passed in. If you want to save the results of this function, you need to assign them to a variable.

```
>>> a = [2, 1]
>>> sorted(a)
[1, 2]
>>> a
[2, 1]
>>> a = sorted(a)
>>> a
[1, 2]
```

BUT, some functions operate on data “in place”, or “mutate” it. Note that since a and b refer to the same thing, it changes both.

```
>>> a = [1, 2]
>>> b = a
>>> a.append(3)
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
```

With that established, some things worth mentioning:

Typically functions that mutate an object don't return anything.

```
>>> b = a.append(3)
>>> print b
None
```

Strings are immutable. There are no string functions that change a string in place- they always create new ones.

```
>>> a = “hello world”
>>> a.split()
[‘hello’, ‘world’]
>>> a
‘hello world’
```

Tuples are like lists but they're immutable. This means they're great for things that should never change in length or in relationship to each other (like pairs or groups of things that belong together)

```
>>> a = (1,2)
>>> a[1] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```