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.
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!
How Python works: Namespaces
Some vocabulary

We’ll use the terms “value” and “object” interchangeably. We’ll use the terms “name” and “variable” interchangeably.

A binding is a runtime pair: name ↦ value.

A namespace is a runtime collection of bindings.

At runtime, an assignment binds a name to a value.

At runtime, a reference looks up a name’s value.

A name’s scope is the region of text in which that name is valid.
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

**scopes**

(determined by program code)
global / file / module / session

```python
x, y = 'a', 'b'
def f1():
    x = 1
    print(x, y)
def f2(y):
    x = 2
    print(x, y)
f1()
f2(3)
print(type(x), type(y))
print(x, y)
```

scopes
(determined by program code)

namespaces
(a snapshot of program execution)
Built-in

global / file / module / session

1  x, y = 'a', 'b'
2
3  def f1():
   4     x = 1
   5     print(x, y)
   6
   7  def f2(y):
   8      x = 2
   9      print(x, y)
  10
 11  f1()
 12  f2(3)
 13  print(type(x), type(y))
 14  print(x, y)

Scopes
(determined by program code)

Namespaces
(a snapshot of program execution)
built-in

global / file / module / session

```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

f1()

def f2(y):
    x = 2
    print(x, y)

f2(3)

print(type(x), type(y))
print(x, y)
```

scopes
(determined by program code)

namespaces
(a snapshot of program execution)
```python
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
```python
x, y = 'a', 'b'
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
```python
def f1():
    x = 1
    print(x, y)

f1()
def f2(y):
    x = 2
    print(x, y)
f2(3)
```

**Built-in names and scopes**

- Global: `x`, `y` = 'a', 'b'
- Local: `x` = 1, `y` = 'b'
- Built-in: `print`

**Namespaces**

- Global:
  - `x` = 'a'
  - `y` = 'b'
- Local:
  - `x` = 1

**Execution**

- `f1`, called @ line 11
- `f2`, called @ line 13

**Output**

- `print(type(x), type(y))` results in `<class 'int'>, <class 'str'>`
- `print(x, y)` results in `1 'a'`

```python
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
```python
# global / file / module / session
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)
    f1()

def f2(y):
    x = 2
    print(x, y)
    f2(3)

print(type(x), type(y))
print(x, y)
```

Scopes: (determined by program code)

Namespaces: (a snapshot of program execution)
```python
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)

print(type(x), type(y))
print(x, y)
```python
def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
```
x, y = 'a', 'b'

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
Let’s practice!

```python
1  def fact(n):
2      if (n == 0):
3          return 1
4      return n * fact(n-1)

5  n = 4
6  result = fact(n / 2)
```
Modules are just more namespaces.
import functions
functions.f1()

def f1():
    x = 1
    print(x, y)

def f2(y):
    x = 2
    print(x, y)

f1()
f2(3)
print(type(x), type(y))
print(x, y)
import functions
functions.f1()
Slicing a sequence

\[ \text{seq[start : end : step]} \]

think: \([start, end)\)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```python
>>> values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> values[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> values[3:5]
[3, 4]
>>> values[-1]
9
>>> values[::2]
[0, 2, 4, 6, 8]
```
Functional programming in Python
Reading from files

```
open('data.txt').readlines()
['0,1,2,3,4,5,6,7,8,9
', '0,2,4,6,8
', '100,200,300,400,500,600,700,800,900
', '10,17,24,31,38,45,52,59,66,73,80,87,94
']
```

```
open('data.txt').read()
'0,1,2,3,4,5,6,7,8,9
0,2,4,6,8
100,200,300,400,500,600,700,800,900
10,17,24,31,38,45,52,59,66,73,80,87,94
'
```

```
open('data.txt').readlines()
['0,1,2,3,4,5,6,7,8,9
', '0,2,4,6,8
', '100,200,300,400,500,600,700,800,900
', '10,17,24,31,38,45,52,59,66,73,80,87,94
']
```
List comprehensions
are syntactic sugar for functional programming concepts (e.g., `map`)

```python
lines = open('data.txt').readlines()

data = []
for line in lines:
    data.append(line[:-1])

data = list(map(lambda line: line[:-1], lines))

data = [line[:-1] for line in lines]
```

loop (imperative programming)
map (functional programming)
list comprehension (functional programming)
List comprehensions

are syntactic sugar for functional programming concepts (e.g., filter)

```python
positiveValues = []
for value in values:
    if value > 0:
        positiveValues.append(value)

data = list(filter(lambda value: value > 0, values))

data = [value for value in values if value > 0]
```
Use list comprehensions.

List comprehensions are usually clearer (to Python programmers) than `map` / `filter` or single loops that build up lists.
Python environment: VSCode + iPython

```python
def fact(n):
    if (n == 0):
        return 1
    return n * fact(n-1)
```

Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)

Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 — An enhanced Interactive Python. Type '?' for help.

```
In [1]: cd Desktop
/Users/ben/Desktop

In [2]: 
```
Python sounds good!

http://tinyurl.com/hmc-python-sounds

Help with the terminal: http://tinyurl.com/hmc-ipython-terminal

Try to get as far as: replace_some

We’ll stop at 10:45.