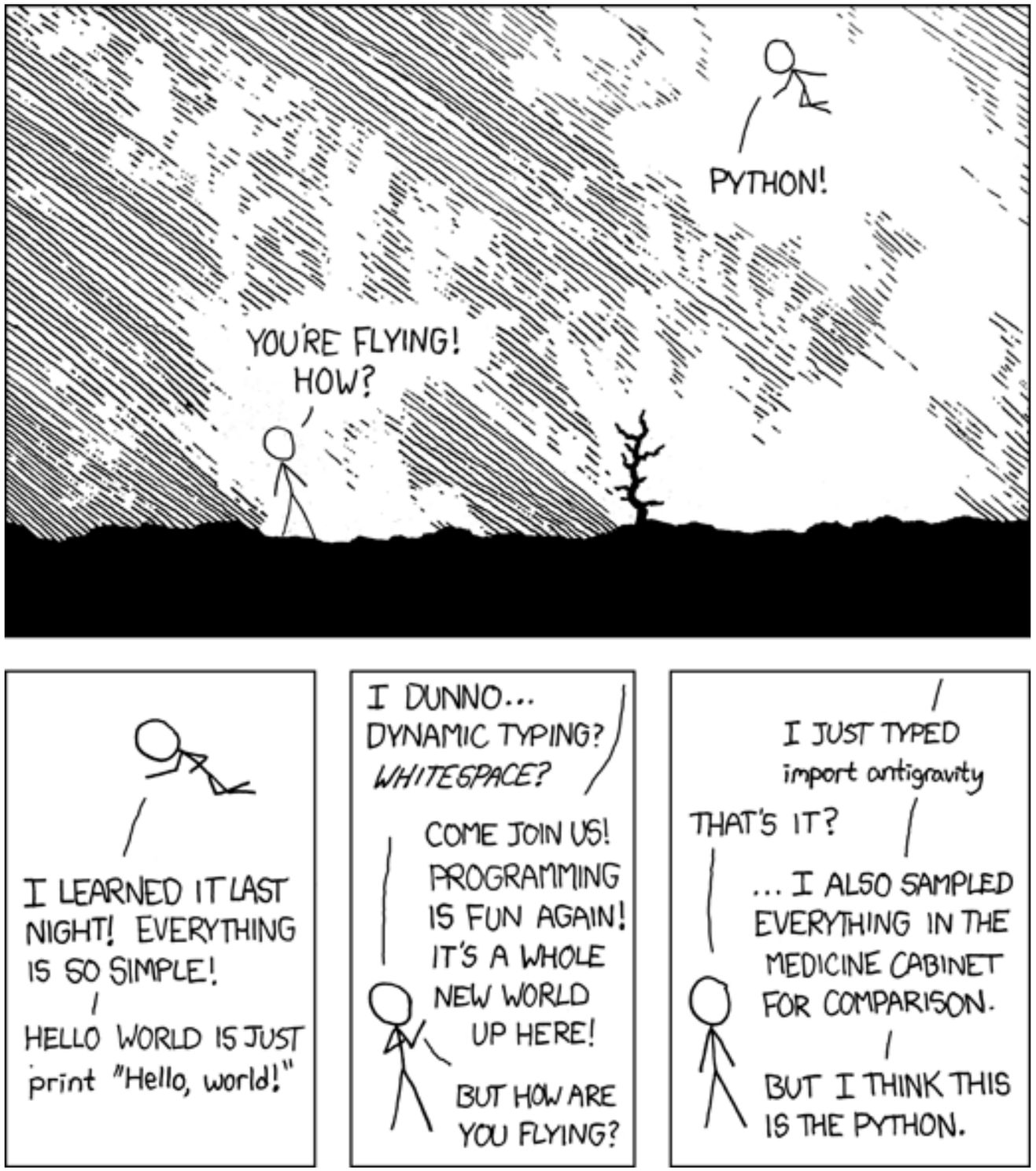
Python - A Crash Course

Antonio Lima

Networks and Distributed Systems 2012-2013
School of Computer Science - University of Birmingham
8 March 2013


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

What is Python?

Python is a an interpreted, general-purpose high-level programming language whose design philosophy empashises code readability.

"(Wikipedia)

- Blocks are identified by indentation (we need no stinkin' { }, code is beautiful)
  - CAREFUL! This means indentation is not optional, it has semantic meaning.
- Multiple programming paradigms: mainly object-oriented and imperative, but also functional. You can write classes or scripts.
- Batteries included: most primitive structures are already implemented: lists, tuples, dictionaries, sets, ... Very rich standard library (networks, json, asynchronous calls, zip files, ...).
- If you need something but you don't find it the standard library, chances are it is somewhere else (PyPI, GitHub, ...). If not, write it yourself, it's fun.

A Programming ~~Language~~ Philosophy

The Zen of Python

```
>>> import this

    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!
```

Getting started

The first thing you need to do (after installation) is launch the python interpreter.

```
$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
```

You can either use the interpreter interactively, "on the fly", or launch a script.

```
$ python script_foo.py
```

Hello world, Dynamic typing

```
>>> print "Hello world"
Hello world
```

No more forgotten ; at the end of lines.

```
# Whatever is on the right of a hash is a comment
>>> a = 3 # int
>>> a = "Hello!" # string, try this in Java...
```

You don’t need to specify types. Types are **evaluated** at runtime.

Wonderful, huh? Not perfect, this makes runtime errors much more common. You need to be more careful with your code.

Built-in types: boolean and numeric types

```
a = True # bool
b = False # bool


a = 3 # int
b = 3.0 # float
c = 0L # long
d = complex(re, im) # complex
```

Built-in types: lists

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = [1, "Sam", True] # Can be non-homogeneous
>>> max(a)
6
>>> min(a)
0
>>> sum(a)
21
>>> len(a)
6
```

List methods

```
>>> a.append(99) # Append an element
>>> a.remove(51) # Remove the first occurrence of 51
>>> a.insert(10, "HI") # Insert "HI" in position 10
>>> a.count(99) # Count the occurrences
>>> a.reverse()
```

Explore all methods interactively using `dir(a)` or `dir(list)`. You can find out what a method does by using

```
>>> help(a.reverse)
```

```
count(...)
    L.count(value) -> integer -- return number of
occurrences of value
```

List slicing, list comprehension

```
>>> a = range(100)
>>> a[:5] # First 5 elements
>>> a[:-5] # Last 5 elements
>>> a[:20:2] # First 5 elements, taken every three
```

```
>>> a.extend(range(100, 200))
```

```
>>> even = [el for el in a if el % 2 == 0]
```

Built-in types: strings

```
>>> a = "I’m a string"
>>> b = ‘I\’m also a string’ # Note the escape character
>>> c = """I’m also a string""" # Triple quotes
>>> a[2]
‘m’
```

```
>>> fruits = "Apple,Strawberry,Watermelon"
>>> fruitlist = fruits.split(",")
['Apple', 'Strawberry', 'Watermelon']
>>> print ‘:‘.join(fruitlist) # Change separator
Apple:Strawberry:Watermelon
>>> fruits.lower() # (or fruits.upper())
"apple,strawberry,watermelon"
```

Built-in types: sequences

```
>>> a = [1, 2, 3, 4]
>>> a[2] = 10
>>> a
[1, 2, 10, 4]
```

Lists are mutable, tuples are immutable.

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> a[2] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Built-in types: sets

Sets are **unordered** collections of **unique** elements. Use them you want to make sure there are no duplicates and when you don't care about the order.

```
>>> a = set()
>>> a.add(1)
>>> a.add(1)
>>> print a
set([1])

>>> list_with_repetitions = [1, 2, 3, 3, 3, 2, 2, 2]
>>> set(list_with_repetitions) # Eliminate duplicates
set([1, 2, 3])
```

Built-in types: sets

Sets allow you to check **quickly** if an element is present.

```
>>> r = range(200000) # Suppose this is a very big
>>> rs = set(r) # Set of the same elements
>>> -3 in r
False
>>> -3 in rs # This operation is much faster, O(1)
False
```

Built-in types: dictionaries

Dictionaries are **unordered** data structures that map keys to values.

```
>>> users_age = {}
>>> users_age["John"] = 24
>>> users_age["Alice"] = 22
>>> users_age["Alice"] = 20
>>> print users_age["Alice"]
20
>>> print users_age.keys()
["Alice", "John"]
>>> print users_age.values()
[24, 22]
```

Interactive input

```
>>> name = raw_input("What is your name?")
>>> age = raw_input("What is your age?")
>>> print "You are", name, "and you are", age
You are Mike and you are 25.
>>> print "In one year you will be", age+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
'str'


>>> print "In one year you will be", int(age) + 1
In one your you will be 26.
```

If

```
people = set(["Alice", "John", "Laura"])
users_ages = {      # Initialize a dictionary
    "Alice": 20,
    "John": 24,
    "Laura": 30,
}
if "Alice" in people:
    print "Present"
if "Alice" in users_ages:
    print "Alice age is", users_ages["Alice"]
```

For

In python, for-loops iterate over sequences.

```
for i in people:
    print i
```

Python is about readability. Choose names wisely.

```
for person in people:
    print person

# Count until 100
for i in range(100):
    print i
```

While

```
i = 0
while i < 20:
    print i
    i += 1
```

It works, but not very **pythonic** ("There should be one-- and preferably only one --obvious way to do it", readability).

```
for i in range(20):
    print i
```

This looks better. Don’t use while for this.

While

Use while when you are not sure of how many iterations you are going to have (for example because they depend on user input or other events).

```
def annoying_question():
    while answer.lower() != "yes"
        print "Are you tired?"
print "Then I'll stop"
```

Lists vs generators

Let’s try to count to a huge number using lists.

```
>>> for i in range(100000000000):
...     print i
# Don’t try, it will hang you will have to kill the process
# Any idea on why?

>>> for i in xrange(100000000000):
...     print i
# This doesn’t hang, it evaluates next at each iteration.
```

Functions

```
def multiply_eight(n):
      return n*8


>>> print multiply_eight(2)
16
>>> print multiply_eight("buffalo") # What do you expect?
```

(Yes, it’s a grammatically valid sentence in American English.)

Functions

Functions can also be passed as a parameter.

```
def apply_function(f, sequence):
    for element in sequence:
        print f(element)
```

```
>>> apply_function(multiply_eight, xrange(3))
0
8
16
```

Classes

```
class Basket(object):
    def __init__(self):  # Initializer
        self.content = []

    def add_element(self, element):
        self.content.append(element)

b = Basket() # Instantiate a new object
b.add_element(3)
```

Modules

A module is a file containing Python definitions and statements (variables, functions, classes, ...). You can reference a module from other modules in the same directory or in the `sys.path` by using `import`. `sys.path` contains.

```
import basket
b = basket.Basket()
```

You can also import all items in the current namespace. However, it is not suggested (name pollution).

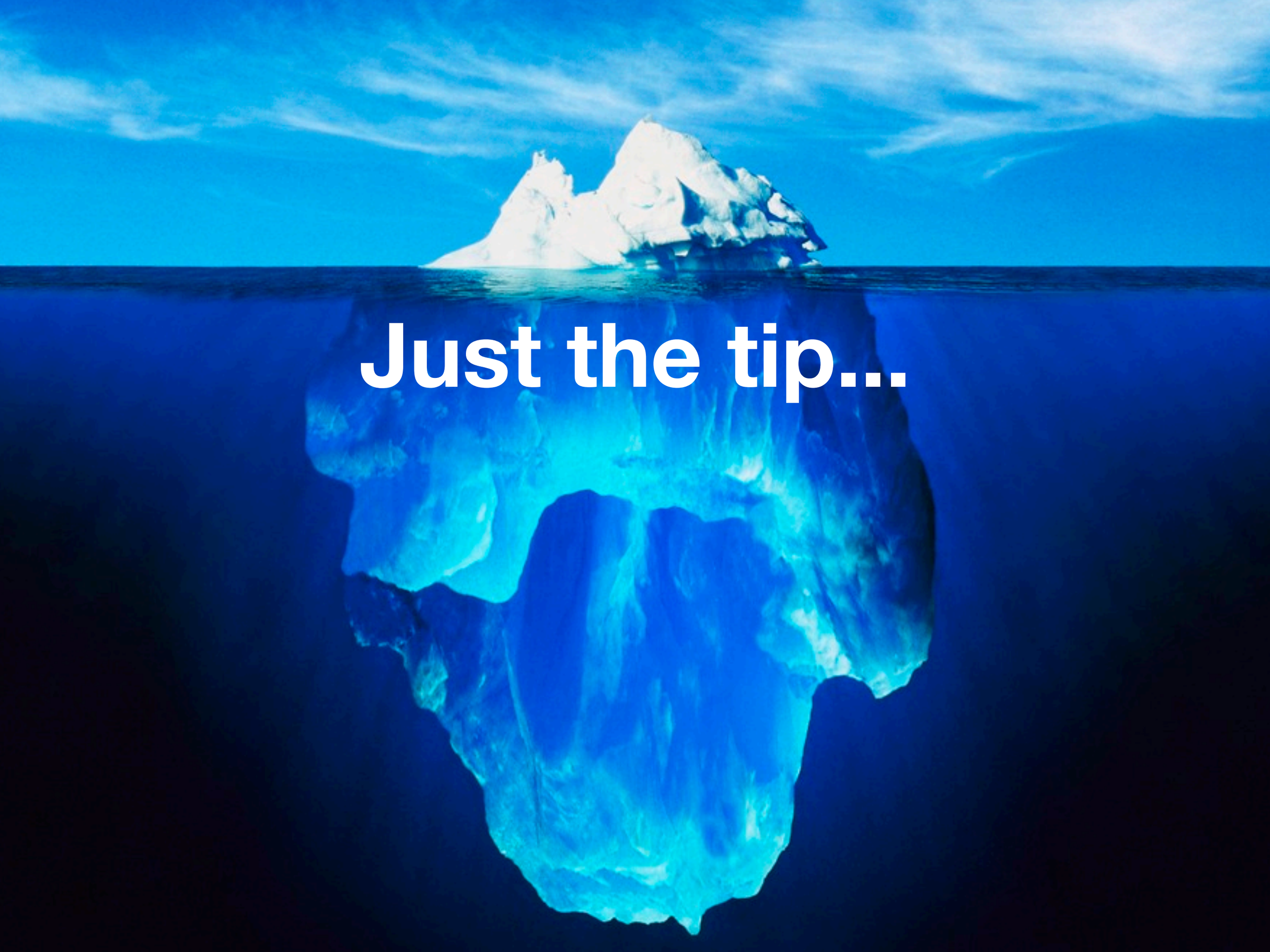
```
from basket import *
b = Basket()
```

Files

```
f = open("foo.txt", "w+")
f.write("Hello!\n")
f.close()
```

Don’t forget to close after writing. It is suggested to use the following.

```
with open("foo.txt", "w+") as f:
      f.write("Hello!\n")
# Closes automatically
```


Just the tip...