Python

Simply Beautiful

Val Neekman

History

"Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C."

"Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library."

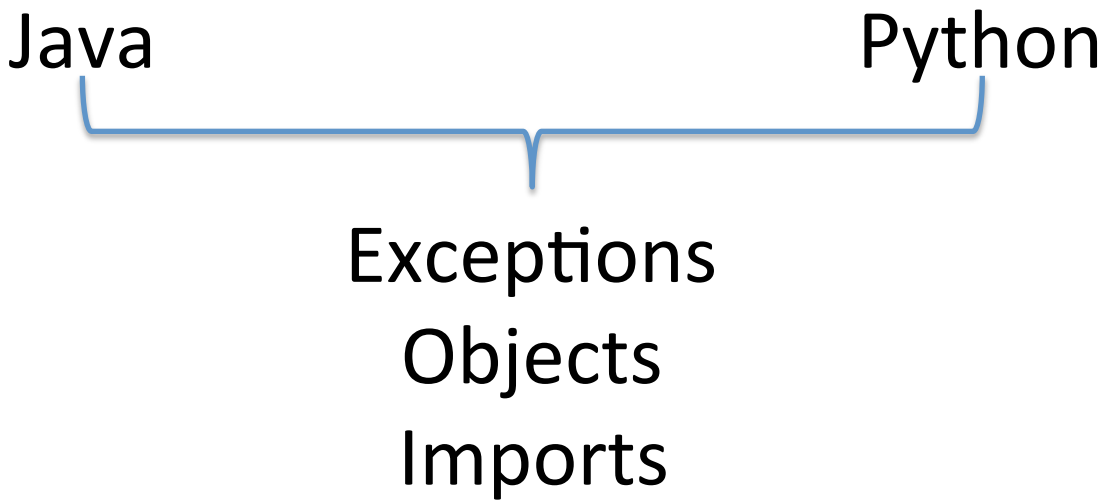
Wikipedia

Audience

You have already been exposed to at least one other programming language

Python – Is it radically different?

- It looks much like several earlier languages



C++ ←→ Python

```
1  // some comment here
2  if ((number < 0.0) && ( mode == 'a'))
3    value = value + number;
4
```

```
1  # some comment here
2  if number < 0.0 and mode == 'a':
3    value = value + number
4
```

Data Structures

- List [ ]
  - The most used data structure in Python. It is versatile and can contain different data types.
  - It is mutable with indices start from zero. It can be sliced, concatenated ..etc.
  - Its closest relative is the “string” which unlike list is immutable.
- Tuple ( )
  - It is immutable and is surrounded by parentheses
  - However, it can hold mutable data (e.g. list and dict)
- Dictionary { }
  - Is an associative collections represented by key:value
  - Keys must be hashable and unique
  - Data can be of any im/mutable types
- Set { }
  - Is an unordered collection of unique values
  - All elements in a set must be hashable (hint: immutable elements)

Data Structures – List [‘when’, 2, ’use’]

- The data doesn't have to be unique.
  - L = [1, 1, 2, 3, 4, 5, 5, 1]
- A single collection of mixed data.
  - L = ['a', 1, 2.0, (1,0), [2,3,], {1,2}, {1: 2}]
- The data order is important.
  - L = [1, 1, 1, 2, 3, 4, 5, 5]
- The ability to change or extend data is required.
  - append(), insert(), remove(), pop() … etc.
- A stack or a queue is required.
  - appending/removing elements from the beginning/end -- append() vs pop().

Data Structures – Tuple (‘when’, 2, ’use’)

- The data doesn't have to be unique.
  - T = (1, 1, 2, 3, 4, 5, 5, 1)
- A single collection of mixed data.
  - T = ('a', 1, 2.0, (1,0), [2,3,], {1,2}, {1: 2})
- The data order is important.
  - T = (1, 1, 1, 2, 3, 4, 5, 5)
- The data needs to be read only. (e.g. configuration data)
  - Immutable (read only)

Data Structures – Dictionary {'when': '2-use'}

- The data requires a logical association between key:value
  - D = {1: 'a', 2: 'b', 3: 'c',}
- A single collection of mixed data with unique keys
  - D = {1: 'a', (1,2): 'b', 'foo': 'c', 2.0: 'two',}
- The data order is not important.
  - Order is not guaranteed
  - 3rd party collections (OrderedDict) library can be used. (Performance?)
- The data can change often
  - Mutable

Data Structures – Set {‘when’, ’2’, ’use’}

- A unique set of data is required.
  - S = {1, 2, 3, 4, 5}
- A single collection of mixed data is required.
  - S = {'a', 1, 2.0, (1,0))
- The data order is important.
  - S = {1, 3, 4, 5, 2}
- The data can change often
  - Mutable
  - Use frozenset() for immutable (read only)
- The data can be manipulated mathematically
  - Difference, union, intersection, symmetric difference.

Python Class -- Definition

- A class is a blueprint/template for creating extensible objects (hint: object-oriented)
  - A way of aggregating similar data and functions
  - An essential data / functionally abstraction
  - Allows for creating maintainable complex programs
- Has a set of default constructor /destructor
  - __init__(), __del__()
- An object created by the class constructor is called an instance of that class

Python Class -- Example

```
1  class PersonClass:
2      """A simple example class"""
3      id = 12345
4      name = "Mike"
5      def person(self):
6          print 'Hello {}'.format(self.name)
```

Python Class -- Polymorphism

Poly (many)
Morph (form/shape)

Python Class -- Polymorphism

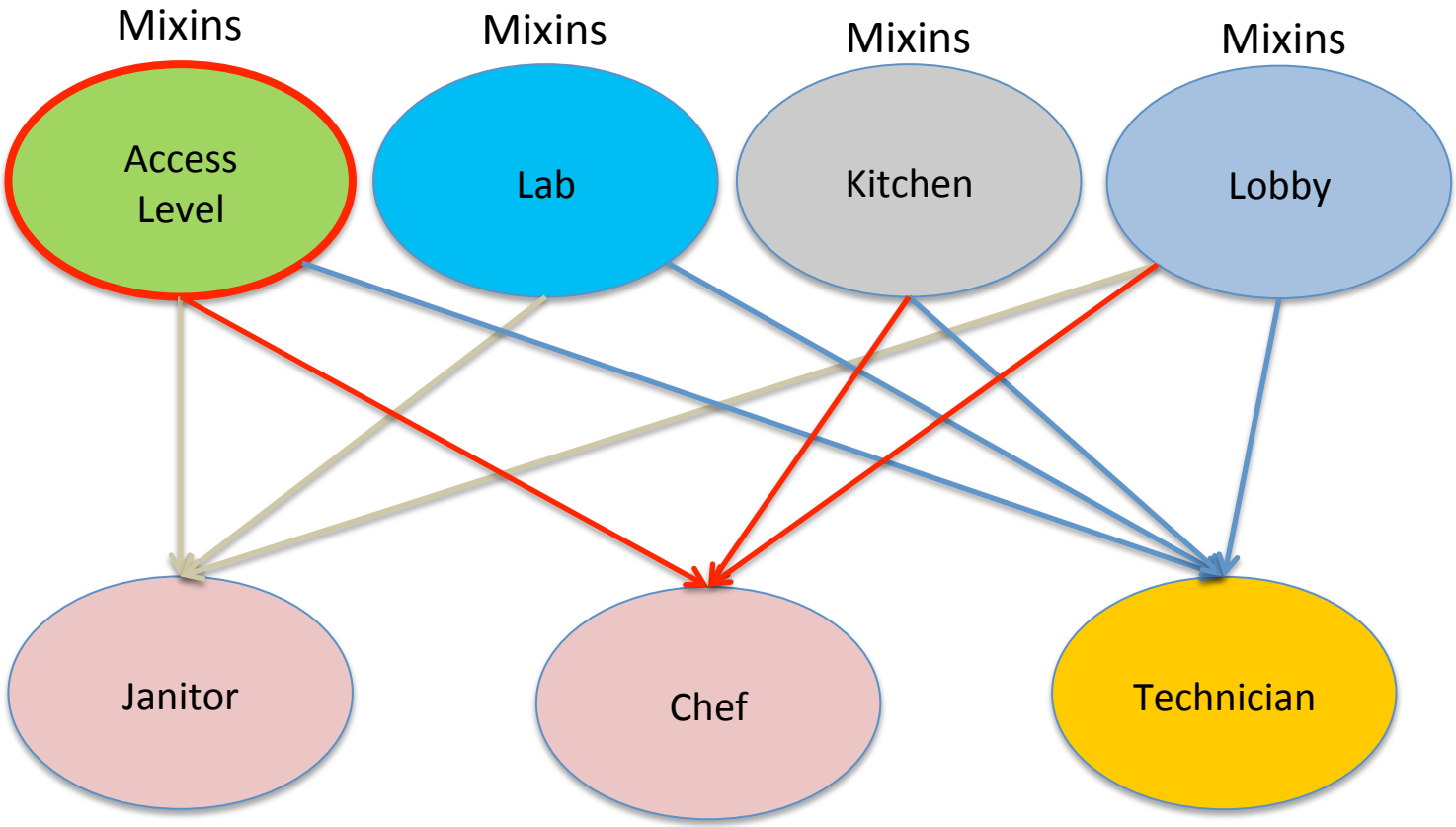
```
1  class Developer(object):
2      def remind(self):
3          raise NotImplementedError
4
5  class PythonDeveloper(Developer):
6      def remind(self):
7          print "In Python, don't forget your indentations."
8
9  class JavaDeveloper(Developer):
10     def remind(self):
11         print "In Java, don't forget your semicolons."
12
13 p = PythonDeveloper()
14 p.remind()
15
16 j = JavaDeveloper()
17 j.remind()
```

The ability to change form base on input/context

Python Class -- Mixins

Mixins in Python, are classes that allow a set of clearly defined methods packaged together into a single in/dependent cohesive unit to be used to add specific functionalities to other classes by “mixing” them in.

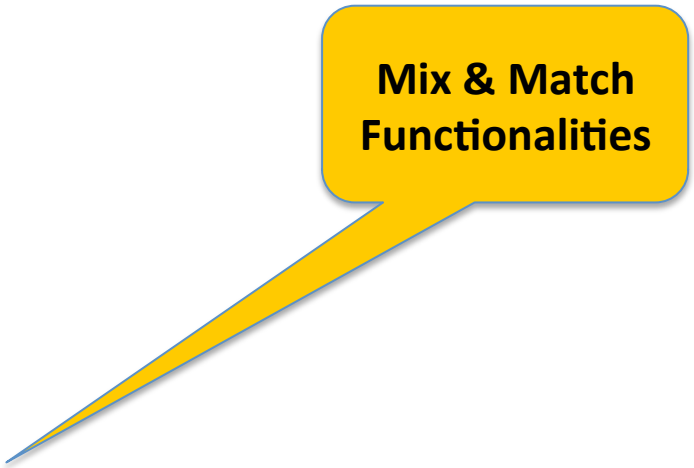
Python Class -- Mixins



Mix & Match Functionalities

Python Class -- Mixins

```
1  class Auth(object):
2      name = "BaseAuth"
3      logged_in = False
4      def is_user_authenticated(self):
5          return self.logged_in
6
7  class OAuth1(object):
8      def three_legged_auth(self):
9          return call_provider()
10
11 class OAuth2(object):
12     def two_legged_auth(self):
13         return call_provider()
14
15 class Google(Auth, OAuth2):
16     name = "Google"
17     def login(self):
18         logged_in = self.two_legged_auth()
19
20 class Yahoo(Auth, OAuth1):
21     name = "Yahoo"
22     def login(self):
23         logged_in = self.three_legged_auth()
24
```



Python -- Decorator

Wrap a function in another function

```
1  def p_decorate(func):
2      def func_wrapper(name):
3          return "<p>{0}</p>".format(func(name))
4      return func_wrapper
5
6  def strong_decorate(func):
7      def func_wrapper(name):
8          return "<strong>{0}</strong>".format(func(name))
9      return func_wrapper
10
11 def div_decorate(func):
12     def func_wrapper(name):
13         return "<div>{0}</div>".format(func(name))
14     return func_wrapper
```

Python – Decorator Usage

```
@p_decorate
@div_decorate
@strong_decorate
def get_text(name):
    return "{0}, you can do it".format(name)

# OR

get_text = p_decorate(div_decorate(strong_decorate(get_text)))


print get_text("Mike")

# Outputs
<p><div><strong>Mike, you can do it.</strong></div></p>
```

Python - Profiler

See how your program is performing. Find the bottleneck.

```
1  import cProfile
2
3  def profileit(func):
4      """
5          Decorator (function wrapper) that profiles a single function
6
7          @profileit()
8          def func1(...)
9              # do something
10             pass
11     """
12     def wrapper(*args, **kwargs):
13         func_name = func.__name__ + ".pfl"
14         prof = cProfile.Profile()
15         retval = prof.runcall(func, *args, **kwargs)
16         prof.dump_stats(func_name)
17         return retval
18
19     return wrapper
```

Python – Profiler Usage

```
# Example
@profileit
def foo():
    a = 0
    for b in range(1, 100000):
        a += b

foo()

# Analysis
#python -m pstats foo.pfl
#Welcome to the profile statistics browser.
#foo.pfl% help
#foo.pfl% sort time
#foo.pfl% stats 10
#Fri Mar 14 10:26:03 2014    foo.pfl
#
#         7 function calls in 0.008 seconds
#
#   Ordered by: internal time
#
#   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
#        1    0.007    0.007    0.007    0.007 {method 'enable' of '_lsprof.Profiler' objects}
#        1    0.002    0.002    0.008    0.008 profile.py:1(<module>)
#        1    0.000    0.000    0.000    0.000 C:\Python27\lib\cProfile.py:5(<module>)
#        1    0.000    0.000    0.007    0.007 profile.py:12(wrapper)
#        1    0.000    0.000    0.007    0.007 C:\Python27\lib\cProfile.py:146(runcall)
#        1    0.000    0.000    0.000    0.000 C:\Python27\lib\cProfile.py:66(Profile)
#        1    0.000    0.000    0.000    0.000 profile.py:3(profileit)
```

Python -- Unittest

```python
import unittest

class MyTests(unittest.TestCase):

    def setUp(self):
        """
        This method is called before each test
        """
        self.mylist = [1,2,3]

    def test_not_equal(self):
        self.assertNotEqual(1, "1")

    def test_equal(self):
        self.assertEqual(1, int("1"))

    def test_true(self):
        self.assertTrue( 3 == len(self.mylist))

    def test_false(self):
        self.assertFalse( 5 == len(self.mylist))

    def test_raise(self):
        self.assertRaises(IndexError, lambda: self.mylist[4])

    def tearDown(self):
        """
        This method is called after each test
        """
        pass

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

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Test what you ship
Ship what you test