Programming with Python

Adv. Topics

Mosky

Mosky:

- The examples and the PDF version are available at:
  - j.mp/mosky-programming-with-python.
- It is welcome to give me any advice of this slide or ask me the answers of the challenges.
  - mosky.tw

Topics

- Basic Topics
  - Python 2 or 3?
  - Environment
  - hello.py
  - Common Types
  - Flow Control
  - File I/O
  - Documentation
  - Scope
- Adv. Topics
  - Module and Package
  - Typing
  - Comprehension
  - Functional Technique
  - Object-oriented Prog.
  - Useful Libraries
- Final Project
  - A Blog System

Module and Package

Write you own module and package!

Module and Package

- A Python file is just a Python module:
  - `import module_a # module_a.py`
- A folder which has `__init__.py` is just a Python package:
  - `import package_x # __init__.py`
  - `import package_x.module_b # package_x/module_b.py`
  - `from . import module_c`
    `# (in package_x.moudle_b) package_x/module_c.py`
  - `$ python -m package_x.module_b`
- Do *not* name your file as any built-in module.
  - ex. `sys.py`

Module and Package (cont.)

- The tree:

```
- .
   ├── moudle_a.py
   └── package_x
       ├── __init__.py
       ├── module_b.py
       └── module_c.py
```

The `import` Statement

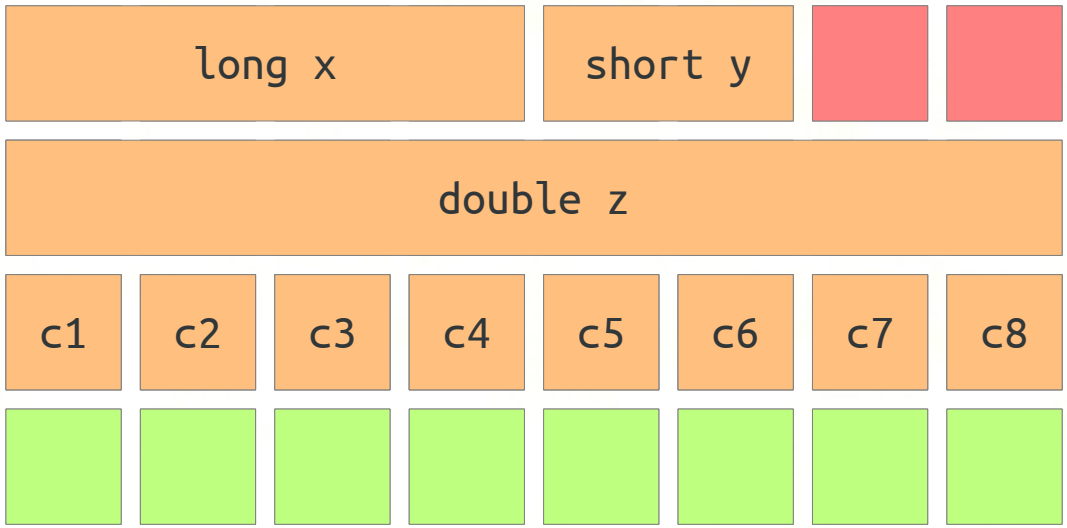
- A module is only imported at the first import.
- ```
  import module
  module.val = 'modified'
  ```
  - The module is affected by this modification.
- ```
  from module import val
  val = 'modified'
  ```
  - The module is *not* affected by this modification.
  - It does a *shallow* copy.

Typing

static? dynamic? weak? strong?

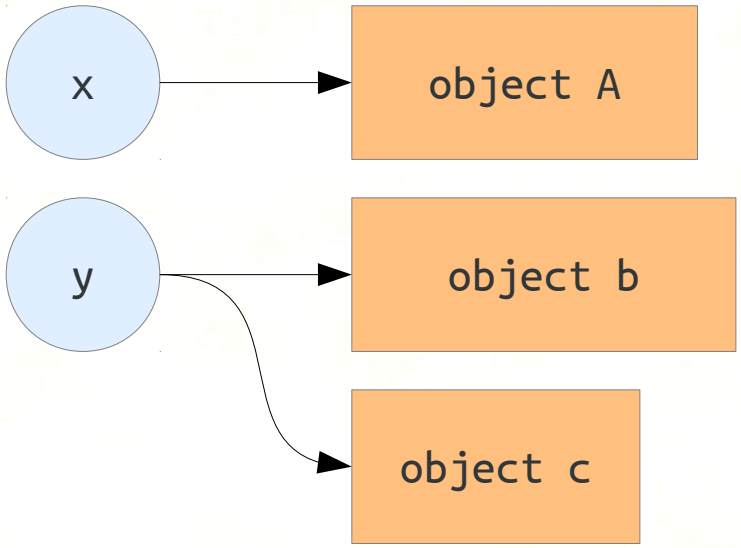
Static Typing

- Checking types in *compile time*.
- Usually, it is required to give a type to a variable.
- Python **is not** static typing.



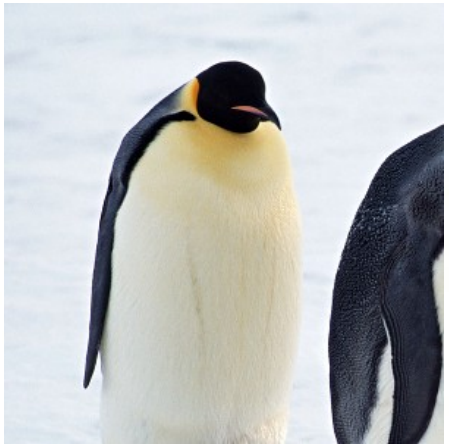
Dynamic Typing

- Checking types in *run time*.
- A variable just points to an object.
- Python **is** dynamic typing.



NOTE: This is an animation and it is not correct in the PDF version.

Duck Typing



Duck Typing (cont.)

- A style of dynamic typing.
- Happy coding without the *template*, the *generics* … etc.
- If it is necessary to check type:
  - `if hasattr(x, '__iter__'):`
    - adapt the type inputed
  - `assert not hasattr(x, '__iter__'), 'x must be iterable'`
    - notify the programmer
  - `if isinstance(x, basestring):`
    - the worst choice

Duck Typing (cont.)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# file: ex_dyn.py

def dynsum(*seq):

    r = seq[0]
    for item in seq[1:]:
        r += item

    return r

if __name__ == '__main__':

    print dynsum(1, 2, 3)
    print dynsum('x', 'y', 'z')
```

- String and integer both support += operator.
- Write the code with elasticity.

Duck Typing (cont.)

- BUT, it will confuse you when your project is going to big.
  - *Name* your variables with hint of type.
    - `item` vs. `items`
    - `employee` vs. `employee_name`
    - `args` vs. `kargs`
  - *Documentation* does matter.

Weak Typing

- It converts the type if you do an operation which is not supported by the original type.
- In JavaScript:
  - `1 + '1'`
    → `'11'`
  - `1 == '1'`
    → `true`
- Python **is not** weak typing!

Strong Typing

- Only do the operations which are supported by the original type.
  - `1 + '1'`
    - → `TypeError`
  - `1 == '1'`
    - → `False`
- Python **is** strong typing!

Comprehension

Compact your statements.

List Comprehension

```
[i for i in range(10)]
[i ** 2 for i in range(10)]
[f(i) for i in range(10)]
[i for i in range(10) if i % 2 == 0]
[i for i in range(10) if not i % 2 == 0]
[i for i in range(10) if g(i)]
```

List Comprehension (cont.)

List comprehension:

```
[
    (i, j)
    for i in range(3)
    for j in range(3)
]
```

is equal to:

```
r = []
for i in range(3):
    for j in range(3):
        r.append((i, j))
```

List Comprehension (cont.)

List comprehension:

```
[
    [
        (i, j)
        for i in range(3)
    ]
    for j in range(3)
]
```

is equal to:

```
r = []
for i in range(3):
    t = []
    for j in range(3):
        t.append((i, j))
    r.append(t)
```

Generator Comprehension

- Generator comprehension:
  - The examples:
    - `(i for i in range(10))`
    - `f(i for i in range(10))`
  - It is like `xrange`.
  - Lazy evaluation → Save memory.

Other Comprehensions

Python 3 only:

- set comprehension:
  - `{i for i in range(10)}`
- dict comprehension:
  - `{i:i for i in range(10)}`

But we can do so with below statements:

- set comprehension:
  - `set(i for i in range(10))`
- dict comprehension:
  - `dict((i, i) for i in range(10))`

Functional Technique

Think in the functional way.

The any/all Function

- ```
  def all_even(seq):
      return all(i % 2 == 0 for i in seq)
  ```
- ```
  all(type(item) is int for item in inputs)
  ```
- ```
  any(i < 0 for i in inputs)
  ```
  ```

Challenge 3-3: The Primes (cont.)

- limit: in one line.
  - hint: use `any` or `all`

```
[2, 3, 5, 7, 11, 13,
17, 19, 23, 29, 31,
37, 41, 43, 47, 53,
59, 61, 67, 71, 73,
79, 83, 89, 97]
```

The `zip` Function

- ```
  matrix = [
      [1, 2],
      [3, 4],
  ]
  ```
- ```
  zip(*matrix)
  ```

The `zip` Function (cont.)

- ```
  result = [
      ('I am A.', 'email_A'),
      ('I am B.', 'email_B'),
  ]
  ```
- ```
  emails = zip(*result)[1]
  ```

First-class Functions

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# file: ex_do.py

from operator import add, mul

def do(action, x, y):
    return action(x, y)

if __name__ == '__main__':
    print do(add, 10, 20)
    print do(mul, 10, 20)
```

- passing functions as arguments.

The `lambda` Expression

- `lambda` [args]`:` [expression]
- It defines an anonymous function.
- It only allows a single expression.
- `f = lambda x: g(x)+h(x)`
- `do(lambda x, y: (x+y)*(x+y), 10, 20)`

Use `sort` with Lambda

```
d = dict(a=300, b=200, c=100)
keys = d.keys()
keys.sort(key=lambda k: d[k])
for k in keys:
    print k, d[k]
```

Use `sort` with Lambda (cont.)

```
names  = ['Andy', 'Bob', 'Cindy']
scores = [70, 100, 95]
table  = zip(names, scores)
table.sort(key=lambda pair: pair[1])
for name, score in table:
    print name, score
```

The map Function

- `map(lambda x: x**2, range(10))`
- `map(int, '1 2 3'.split(' '))`
- `map(ord, 'String')`
- `map(open, [<paths>])`
- `map(str.split, open(<path>))`

The `map` Function (cont.)

- `from operator import mul`
- `a = (1, 2)`
- `b = (3, 4)`
- `sum(map(mul, a, b))`

The `filter` Function

- `filter(lambda i: i % 2 == 0, range(10))`
- `filter(str.isdigit, strings)`
- `filter(lambda s: s.endswith('.py'), file_names)`

Comprehension vs. map/filter

- `[i ** 2 for i in range(10)]`
- `map(lambda i: i ** 2, range(10))`

- `[i ** 2 for i in range(10) if i % 2 == 0]`
-
```
map(lambda i: i ** 2, filter(
    lambda i: i % 2 == 0,
    range(10)
))
```

Comprehension vs. `map`/`filter` (cont.)

- `[ord(c) for c in 'ABC']`
- `map(ord, 'ABC')`

Comprehension vs. `map`/`filter` (cont.)

Compare the speeds of them:

1. `map`/`filter` (with built-in function)
2. comprehension
3. `map`/`filter`

The reduce Function

```
# from functools import reduce # py3
from operator import add


seq = [-1, 0, 1]
reduce(add, s)


seq = ['reduce ', 'the ', 'lines.']
reduce(add, s)
```

The `partial` Function

- `from functools import partial`
- `from operator import add`

- `rdsum = partial(reduce, add)`
- `rdsum([-1, 0, 1])`
- `rdsum(['reduce ', 'the ', 'lines.'])`

The `partial` Function (cont.)

- `from functools import partial`
- `from fractions import gcd as _gcd`

- `_gcd(6, 14)`
- `gcd = partial(reduce, _gcd)`
- `gcd([6, 14, 26])`

Closure

- `from math import log`
- ```
  def mklog(n):
      return lambda x: log(x, n)
  ```
- `log10 = mklog(10)`
- `log10(100) # n = 10`
  ```

Closure (cont.)

```
setattr(DictLike, attrname,
    # it is a colsure
    (lambda x:
        property(
            lambda self: self.__getitem__(x),
            lambda self, v: self.__setitem__(x, v),
            lambda self: self.__delitem__(x)
        )
    )(attrname)
)
```

The `yield` Statement

- ```
  def mkgen(n):
      for i in range(n):
          yield i ** 2
  ```

- It is a generator.

- ```
  gen = mkgen(10)
  ```
- ```
  for i in gen:
      print i
  ```
  ```

Decorator

```
def deco(f):
    def f_wrapper(*args, **kargs):
        print 'DEBUG: ', args, kargs
        return f(*args, **kargs)
    return f_wrapper
```

```
@deco # is equal to add = deco(add)
def add(x, y):
    return x+y
```

```
add(1, 2)
```

Object-oriented Programming

is also available.

The `class` Statement

```
class Example(object):
    class_attribute = 1
    def method(self, …):
        pass
example = Example()
print example
```

The Class in Python (cont.)

- *Everything* in Python is object.
  - Class is an object, too.
- All class inherit the `object` → new-style classes
  - Use new-style classes. It provides more features.
  - Python 3: auto inherit the `object`.
- Supports multiple inheritance.
  - Searching attributes/methods is like BFS.

Bound and Unbound Method

- unbound method
  - `def m(self, ...)`
  - `C.m(c, ...)`
- bound method (instance method)
  - `c.m(...)`

Class Method and Static Method

- class method
  - `@classmethod`
    `def m(cls, …)`
  - `C.m()`
  - `c.m()`
- static method
  - `@staticmethod`
    `def m(...)`
  - `C.m()`
  - `c.m()`

The Data Model of Python

- Special methods
  - `__init__`
  - `__str__`
  - `__repr__`
  - `__getitem__` → `x[key]`
  - `__setitem__` → `x[key] = value`
  - `__delitem__` → `del x[key]`
  - …
- ref: docs.python.org/2/reference/datamodel.html

Protocol

- It like interface, but it is only described in doc.
- The examples:
  - iterator protocol
    - object which supports `__iter__` and `next`
  - readable
    - object which supports `read`
  - ...

The employee class

- see examples/ex_empolyee.py .

Do Math with Classes

- see `examples/ex_do_math_with_classes/` .

Challenge 6: Give a Raise

- Give your employee a raise.
  - without limit
  - limit: prevent modifying `salary` by attribute.
    - hint: use `property`

```
cindy = Empolyee(...)
cindy.add_salary(1000)
print cindy.salary
```

Useful Libraries

```
import antigravity
```

Useful Libraries

The built-in libraies:

- random
- datetime
- re
- hashlib/hmac
- decimal
- glob
- collections
- subprocess
- multiprocessing
- gc
- pickle
- json
- pprint
- gzip
- timeit
- logging
- unitest
- doctest
- pdb
- ...

Useful Libraries (cont.)

The third-party libraries on PyPI:

- Requests – Use it instead of the poor built-in urllib.
- lxml – Do you need to parse HTML? Use it!
- PyYAML – YAML is a the best of data serialization standards.
- PIL – Python Image Library
- NumPy and SciPy – are for mathematics, science, and engineering.
- SymPy – is for symbolic mathematic
- Bottle, Flask or Django – are the web frameworks.
- Sphinx – helps you to create documentation.

...

Challenge 7: Iterable Interger

- Make integer iterable.
  - without limit
  - limit 1: don't use string
  - limit 2: use `collection`
    - hint: use `property`

```
x = IterableInt(10)
for b in x:
    print b
```

```
0
1
0
1
```

Final Project : A Blog System

Final Project : A Blog System

- The cookbook:
  - Flask – A micro web framework.
  - Use `pickle` to store the posts.
  - Optional:
    - A database instead of `pickle`. (ex. MySQL, PostgreSQL, ...)
    - A cache layer. (ex. memcached, redis, ...)
    - A message queue for async jobs. (ex. RabbitMQ, ...)

It is the end.

Contact me? http://mosky.tw/