Imperial College London

Software Engineering Practice

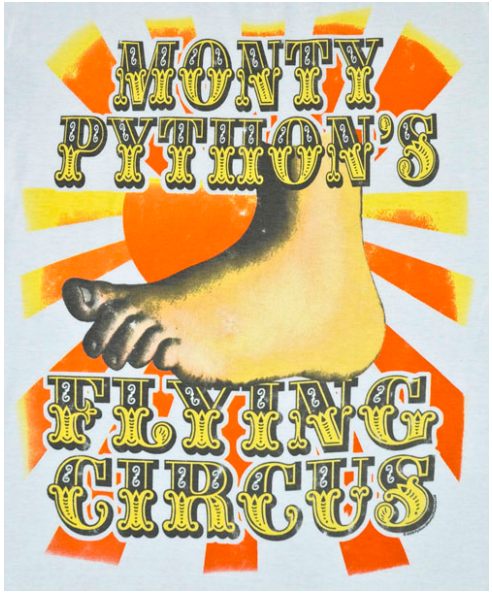
Introduction to Python

A Bit of History

- Developed by Guido Van Rossum in late 80s and first released in early 90s
- Python 2.0 released in 2000
  - Version 2.7 came in 2010
    - Still the “standard” version
    - but is the last release in the series
- Python 3.0 released in 2008
  - Addresses perceived design flaws of earlier versions and “cleans up” the language
  - **Not** backwards compatible

What's In The Name?

- Named after Monty Python's Flying Circus …



- Playful approach ('foo' and 'bar' vs 'spam' and 'eggs')

Python: What Is It?

- Open source, general-purpose programming language
  - Usually referred to as a scripting language
- Multi-paradigm, incorporates elements of:
  - imperative/procedural, object-oriented, and functional styles
- Dynamically typed and garbage collected
- Excellent library support
  - Standard installation comes with a wide range of library support
  - Lots of very useful 3rd party libraries

Python: What Is It?

- Has influenced many subsequent languages
  - E.g. Javascript (ECMAScript), Ruby, Go
- Python code is compiled to bytecode which is then interpreted (just like Java)
  - The reference interpreter is CPython (open source; written in C)
  - PyPy is an interpreter that uses JIT compilation techniques

Python: What Is It?

Can be compiled to other platforms if desired

- allows Python code to interface with other code:
  - Jython compiles Python programs to Java bytecode
  - Iron Python compiles Python to .Net; Common Language Runtime (CLR) bytecode
  - Cython translates Python to C code which is then compiled and can be called by Python code run using CPython
  - py2exe is an extension that converts Python scripts to Windows executables

Running Python Code

- Simply invoke the interpreter on the command line

```
> python file_name.py
```

- Scripts can be made executable in the usual way

```
> cat HelloWorld.py
#!/usr/bin/env python
print("Hello World")

> chmod u+x HelloWorld.py

> ./HelloWorld.py
Hello World
```

Running Python Code

- The interpreter can also be started in *interactive* mode

```
> python
Python 2.7.12 (default, Nov 19 2016, 03:20:26)
[GCC 5.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello World")
Hello World
>>>
```

  - Exit the interpreter by sending EOF character (CTL-D) or exit()
- Note: default installation is version 2, but we can run version 3

```
> python3
Python 3.5.2 (default, Nov 17 2016, 19:29:01)
...
```

Readable, Uncluttered Code (?)

C-style syntax

```
int x = 0;
while (x < 10) {
  int y;
  if (x > 2 && x < 7) {
    for (int z = 0; z < x; z += 2)
      y -= z;
  } else if (x < 2) {
    y = 0;
  } else
    y++;
  print(y);
  x++;
}
```

Python

```
x = 0
while x < 10:
  if (x > 2 and x < 7):
    for z in range(0, x, 2)
      y -= z
  elif x < 2:
    y = 0
  else:
    y += 1
  print(y)
  x += 1
```

Readable, Uncluttered Code (?)

- No need to declare variables and types (e.g. int x)
  - First use adds the variable to the local stack
- No braces: block scope determined by indentation
  - Whitespace is significant
  - Be careful when mixing spaces and tabs!
- Usual control flow statements (if, for, while, etc.)
  - But subtle differences (e.g. in syntax of for loop, elif, etc.)
- Keywords for boolean operators are and, or, not
  - Keywords for arithmetic comparisons the same
- Usual assignment operators
  - but no increment (++) or decrement (--)

About Basic Data Types

- Numbers and strings are immutable (cannot be modified)
- They are actually stored on the heap ("everything is an object")
  - Referenced (pointed to) by local variables
- Numeric values can be integer or floating point
  - In Python 2, division (/) on integers returns an integer (7/3 == 2), and division on mixed values is floating point (7/2.0 == 3.5)
  - In Python 3, the / operator is always floating point division, and the // operator is used to denote integer division
- Integers can have arbitrary size (no defined limit)
- Floating point accuracy is limited (by hardware)
  - But there are library classes for representing arbitrary decimals and fractions

Built-In Data Structures

- Built-in tuple, list, set, and dictionary types
- Tuples and lists are sequence data types
  - Tuples are immutable
  - Lists are mutable
- Sets behave like mathematical sets
  - Mutable
- Dictionaries map keys to values
  - Keys can only be of immutable types
  - Values can be of mutable types

Lists

- Lists in Python are basically like (extendable) arrays
- Elements of lists can be indexed using the `[]` notation
  - Can be indexed both forwards and backwards!
- Subsets of lists can be obtained using the slice (`:`) operator
- Lists respond to methods (append, extend, insert, del, etc.)
- We can create new lists using ***comprehensions***

Lists

```
>>> a = [1, 2, 3, 4, 5]
>>> a[1]
2
>>> a[-1]
5
>>> a[1:4]
[2, 3, 4]
>>> b = a
>>> b.append(6)
>>> b.extend([7, 8, 9])
>>> a.insert(0, 'Peter')
>>> del a[4]
>>> a
['Peter', 1, 2, 3, 5, 6, 7, 8, 9]
>>> [x*x for x in range(1, 10)]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Tuples

- Tuples are sequences of (any kind of) values
- Elements can be indexed using the `[]` notation, like lists
  - Can also be indexed both forwards and backwards
- They are immutable
  - But they can contain mutable elements!
  - Use: constants (because faster than lists)
- Tuples of variables can be involved in assignment
  - Allows multiple variables to be assigned in one go

Tuples

```
>>> t = (1, 2, 'Hello')
>>> t
(1, 2, 'Hello')
>>> u = t, (3.14, 2.71)
>>> u
((1, 2, 'Hello'), (3.14, 2.71))
>>> u[0]
(1, 2, 'Hello')
>>> u[1] = 3
TypeError: 'tuple' object does not support item assignment
>>> v = [1, 2, 3], [4, 5, 6]
>>> v[1].append(7)
>>> v
([1, 2, 3], [4, 5, 6, 7])
>>> w, x, y, z = 1, 'Hello', [2, 3, 4], 9.5
>>> y
[2, 3, 4]
```

Sets

- The set data type implements a mathematical set
  - Unordered collection with no duplicates
  - No element access
  - Mutable (e.g. add, remove, clear)
- Set literals use the notation `{ ... }`
- Sets can be created from lists using the `set()` function
- Sets support the usual set theoretic operations
  - Membership checking (in)
  - Union (|), set difference (-), intersection (&)

Sets

```
>>> s = {1, 2, 3, 4, 1, 5, 2, 6}
>>> s
set([1, 2, 3, 4, 5, 6])
>>>
>>> 3 in s          # set membership
True
>>> 'Hello' in s
False
>>> t = {4, 5, 6, 7, 8, 9}
>>> s - t           # set difference
set([1, 2, 3])
>>> s | t           # set union
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> s & t           # set intersection
set([4, 5, 6])
>>> s ^ t           # difference of union and intersection
set([1, 2, 3, 7, 8, 9])
```

Dictionaries

- The dictionary data type maps keys to values
- Values can be indexed and updated using their corresponding keys with the `[]` notation
  - Keys can be any immutable type (e.g strings, numbers, tuples of immutable objects)
- The dictionary data type is mutable
  - New entries can be added using the `[]` notation
  - Delete and reassign
- We can test whether a dictionary contains a given key using the `in` keyword
- We can obtain `lists` of both the keys and values in a dictionary

Dictionaries

```
>>> d = {'apple' : 'green', 'banana' : 'yellow'}
>>> d['apple']
'green'
>>> d['strawberry'] = 'red'
>>> d
{'apple':'green', 'banana':'yellow', 'strawberry':'red'}
>>> del d['banana']
>>> d['apple'] = 'granny smith'
>>> d
{'apple' : 'granny smith', 'strawberry' : 'red'}
>>> 'apple' in d
True
>>> 'banana' in d
False
>>> list(d.keys()).extend(list(d.values()))
['apple', 'strawberry', 'granny smith', 'red']
```

Defining Functions

- Functions are declared using the `def` keyword

```
def fib(n):      # prints the first n Fibonacci numbers
    a, b = 0, 1
    for i in range(0, n):
        print(a)
        a, b = b, a+b

fib(15)
```

Defining Functions

- Functions arguments can have default values

```
def fib(n = 15):    # prints the first n Fibonacci numbers
    a, b = 0, 1
    for i in range(0, n):
        print(a)
        a, b = b, a+b

fib()
```

Defining Functions

- Functions ***cannot*** be overloaded

```
def fib(n = 15):    # prints the first n Fibonacci numbers
    ...

def fib(start, end):
        # prints the Fibonacci numbers between start and end
    a, b = 0, 1
    while a <= end:
        if a >= start:
            print(a)
        a, b = b, a+b

fib(10)
```

```
TypeError: fib() takes exactly 2 arguments (1 given)
```

Overloading Functions: A Solution

```
def fib(n = None, start = None, end = None):
    a, b = 0, 1
    if start is None and end is None:
        for i in range(0, n):
            print(a)
            a, b = b, a+b
    else:
        if start is not None and end is None:
            end, start = start, n
        while a <= end:
            if a >= start:
                print(a)
            a, b = b, a+b

fib(10)                          #0,1,1,2,3,5,8,13,21,34
fib(5, 10)                       #5,8
fib(start = 10, end = 30)        #13,21
```

Object-Oriented Python

- In Python, we have classes with constructors, fields and methods
  - But all fields and methods are ***public;*** all methods are ***virtual***

```
class Complex:
    def __init__(self, realpart = 0.0, imagpart = 0.0):
        self.r, self.i = realpart, imagpart
        print("Created a new complex number: " + self.displayStr())

    def add(self, other):
        self.r += other.r
        self.i += other.i
        return self

    def displayStr(self):
        return "Complex({:.2f} + {:.2f}i)".format(self.r, self.i)

x = Complex(3.0, 4.0)
x.r, x.i = 5.5, 2.0
print(x.add(Complex(-1.5, 2)).displayStr())
```

Object-Oriented Python

- There is also inheritance and overriding

```
class BetterComplex(Complex):
   def __init__(self, realpart = 0.0, imagpart = 0.0):
      Complex.__init__(self, realpart, imagpart)
      #python 3: super().__init__(realpart, imagpart)

   def double(self):
      self.add(self)
      return self



y = BetterComplex(3.0, 4.0)

print(y.double().displayStr())
```

Exception Handling

- Exceptions can be handled using the `try` keyword

```
try:
    f = open('file.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error")
    raise
```

Exception Handling

- We can define and throw our own exceptions

```
class MyException(Exception):
    def __init__(self, value):
      self.err_value = value

def foo():
    raise MyException(10)

try:
    foo()
except MyException as e:
    print e.err_value
    pass                    #null operation, nothing happens
finally:
    print('This is always printed')
```

Functional Programming in Python

- Functions are '***first class***' entities
  - They can be passed and returned as values

```
def mymap(f):
    def apply_f(l):
        newlist = []
        for elem in l:
            newlist.append(f(elem))
        return newlist
    return apply_f

def square(x):
    return x*x

squarelist = mymap(square)
print squarelist([1, 2, 3, 4])
```

#prints: [1, 4, 9, 16]

Python: Applications / Packages

- Scientific/Mathematical (SciPy, NumPy, Matplotlib, Sage, BioPython, Astropy)
- Web Development (Django, Flask, CherryPy, etc.)
  - Also supported by Google App Engine
- Artificial Intelligence (PyAIML, NLTK)
- Game Development (Pygame)
- Graphics (used as a scripting language for GIMP, Paint Shop Pro, and many others)
- Software Verification (pycontract)
- Logic Programming (pyDatalog)

Further Reading/Practice

- The official Python documentation

  http://www.python.org/doc/

  - Language and library reference, tutorials, etc.

Imperial College London

Questions?