

## Python Programming Introduction

Rabii El Ghorfi

## Introduction

What is Python?

- ▶ Compromise between shell script and C++/Java program
- ▶ Intuitive syntax
- ▶ Interpreted (sort of)
- ▶ Dynamically typed
- ▶ High-level datatypes
- ▶ Module system
- ▶ Just plain awesome

## Introduction

### Java

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

## Introduction

### C++

```
#include <iostream>  
int main()  
{  
    std::cout << "Hello World!" << std::endl;  
    return 0;  
}
```

## Introduction

### Python

```
print "hello world"
```

## Python

- ▶ What does it mean for a language to be “interpreted?”
- ▶ Trick question – “interpreted” and “compiled” refer to implementations, not languages
- ▶ The most common Python implementation (CPython) is a mix of both
  - ▶ Compiles source code to byte code (.pyc files)
  - ▶ Then interprets the byte code directly, executing as it goes
  - ▶ No need to compile to machine language
  - ▶ Essentially, source code can be run directly

## Python

How do you use it?

- ▶ Write code interactively in the interpreter

```
Last login: Wed Jan 15 12:31:56 on ttys004
liidworkin@scas1315:~$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
IGCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.60) on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- ▶ Run a file in the interpreter with `import file`
- ▶ Run a file on the command line with `python file.py`

## Basics

```
>>> 1 + 1
2
>>> print "hello world"
hello world
>>> x = 1
>>> y = 2
>>> x + y
3
>>> print x
1
```

## Types

What does “dynamically typed” mean?

## Types

What does “dynamically typed” mean?

- ▶ Variable types are not declared
- ▶ Python figures the types out at runtime

## Types

- ▶ type function:

```
>>> type(x)
<type 'int'>
```

- ▶ isinstance function:

```
>>> isinstance(x, int)
True
```

- ▶ Difference?

## Types

We prefer to use “duck typing.”

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

— James Whitcomb Riley

```
try:
    # assume object has desired type
except:
    # try something else
```

## Types

What does “strongly typed” mean?

## Types

```
>>> x = 3
>>> x = "hello"
```

- ▶ Has x changed type?
- ▶ No – x is a *name* that *points* to an object
- ▶ First we make an integer object with the value 3 and bind the name 'x' to it
- ▶ Then we make a string object with the value hello, and rebind the name 'x' to it
- ▶ Objects do not change type

## Types

Interpreter keeps track of all types and doesn't allow you to do things that are incompatible with that type:

```
>>> "hi" + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

## Functions

```
def add(x,y):
    return x + y
```

```
>>> add(3,4)
7
```

- ▶ Colon (:) indicates start of a block
- ▶ Following lines are indented

## Types in Functions

- ▶ Function declaration doesn't specify return type
- ▶ But all functions return a value (None if not specified)
- ▶ Parameter datatypes are not specified either

## Style

- ▶ Blocks are denoted by whitespace
- ▶ Use spaces, not tabs
- ▶ Single line comments are denoted with # ...
- ▶ Multi-line comments are denoted with """ ... """
- ▶ Variable and function names should be lower\_case with underscores separating words
- ▶ Use docstrings to document what a function does:

```
def add(x,y):  
    """ Adds two numbers """  
    return x + y
```

## Blocks in the Interpreter

```
>>> def add(x,y):  
...     return x + y  
...  
>>>
```

- ▶ ... indicates more input is expected
- ▶ Need blank line to indicate end of block

## Datatypes: Overview

- ▶ None
- ▶ Booleans (True, False)
- ▶ Integers, Floats
- ▶ Sequences
  - ▶ Lists
  - ▶ Tuples
  - ▶ Strings
  - ▶ Dictionaries
- ▶ Classes and class instances
- ▶ Modules and packages

## Booleans

- ▶ Booleans: True, False
- ▶ The following act like False:
  - ▶ None
  - ▶ 0
  - ▶ Empty sequences
- ▶ Everything else acts like True

## Booleans: Operations

Operation	Result
<code>x or y</code>	if <code>x</code> is false, then <code>y</code> , else <code>x</code>
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>

- ▶ `and`, `or` both return one of their operands
- ▶ `and`, `or` are short-circuit operators

## Booleans: Examples

```
>>> (2 + 4) or False
6
>>> not True
False
>>> not 0
True
>>> 0 and 2
0
>>> True and 7
7
```

## Integers and Floats

- ▶ Numeric operators: `+` `-` `*` `/` `%` `**`
- ▶ No `i++` or `++i`, but we do have `+=` and `-=`
- ▶ Ints vs. Floats

```
>>> int(5/2)
2
>>> 5/2.
2.5
>>> float(5)/2
2.5
>>> int(5.2)
5
```

## Assignments

```
>>> a = b = 0
>>> a, b = 3, 5
Something cool:
>>> a, b = b, a
>>> a
5
>>> b
3
```

## Comparisons

```
>>> 5 == 5
True
>>> "hello" == "hello"
True
>>> 1 != 2
True
>>> 5 > 3
True
>>> "b" > "a"
True
```

## If Statements

```
if a == 0:
    print "a is 0"
elif a == 1:
    print "a is 1"
else:
    print "a is something else"
```

## If Statements

- ▶ Don't need the elif or else
- ▶ Condition can be any value, not just Boolean

```
if 5:
    print "hello"

if "hello":
    print 5
```

## For Loops

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> for i in range(5):
...     print (i)
...
0
1
2
3
4
```

## Ranges

- ▶ `range(n)` produces `[0, 1, ..., n-1]`
- ▶ `range(i, j)` produces `[i, i+1, ..., j-1]`
- ▶ `range(i, j, k)` produces `[i, i+k, ..., m]`

```
>>> range(5, 25, 3)
[5, 8, 11, 14, 17, 20, 23]
```

## Break and Continue

```
>>> for i in range(5):
...     print i
...     if i < 3:
...         continue
...     break
...
0
1
2
3
```

## While Loops

```
>>> i = 0
>>> while i <= 3:
...     print i
...     i += 1
...
0
1
2
3
```



## Example: Factorial Function

$5! = 5*4*3*2*1$   
 $0! = 1$

## Iterative Factorial Function

```
def factorial(x):
```

## Iterative Factorial Function

```
def factorial(x):  
    ans = 1  
    for i in range(2, x+1):  
        ans = ans * i  
    return ans
```

## Recursive Factorial Function

```
def factorial(x):
```

## Recursive Factorial Function

```
def factorial(x):  
    if x == 0:  
        return 1  
    else:  
        return x * factorial(x - 1)
```

## Imports

```
>>> import math  
>>> math.sqrt(9)  
3.0
```

## Python Files

```
import <>  
  
def <>:  
    ...  
  
def <>:  
    ...  
  
def main():  
    ...  
  
if __name__ == "__main__":  
    main()
```

## Python Files

- ▶ `__name__` is a variable that evaluates to the name of the current module
- ▶ e.g. if your file is `h1.py`, `__name__ = 'h1'`
- ▶ But if your code is being run directly, via `python h1.py`, then `__name__ = '__main__'`

## Running Python Files

- ▶ In the **IDLE**:
  - ▶ File open `hello.py`
  - ▶ Run module `F5`
- ▶ In command line:
  - ▶ `python hello.py`