# Introduction to Python

## Summer Projects & Workshop 2017

Ujjawal Soni

Department of Computer Science & Engineering
IIT Madras

June 1, 2017

# Outline

# Why Use Python?

- Software Quality - reusable and maintainable codes
- Developer Productivity - less to type, less to debug, and less to maintain
- Program Portability - most programs run unchanged on all major computer platforms
- Support Libraries - large collection of standard library and third-party extensions
- Component Integration - can integrate with C, C++, Java
- Enjoyment - easy to learn, less efforts

# What Can you do with Python?

- System Programming
  - built-in interfaces to operating-system services
  - python programs can search files and directory trees, launch other programs, do parallel processing with processes and threads, and so on.
- GUI Programming
  - python interfaces like Tkinter allow python programs to implement portable GUIs with a native look and feel
- Internet Scripting
  - can perform wide variety of networking tasks like communicating over sockets, parse web pages
  - packages like Django allows to build quality web pages using python
- Numeric & Scientific Programming
  - python libraries like NumPy, SciPy, TensorFlow support extensive numeric computations

# Running Python Programs

- Using the interpreter:
  - type 'python' on terminal to start the python interpreter.

```
Python 2.7.6 (default, Oct 26 2016, 20:30:19)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Using scripts:
  - save your python statements in a file (say 'program1.py')
  - run the program using - `python program1.py`

# Python Programs Overview

- Programs are composed of modules.
- Modules contain statements.
- Statements contain expressions.
- Expressions create and process objects.

# Comments in Python

There are two commenting styles:

1. Single-line:
   - `# a single-line comment.`
   - `"this is also a single-line comment."`
2. Multi-line:
   - ```
     """
     a multi-line comment.
     any string not assigned to a variable
     is considered a comment.
     """
     ```
     ```

# Variables in Python

- Reserved memory locations to store values.
- Have types like integer, floating-point, character, string, list, etc.
- Unlike C, variables are not declared, just assigned values (using '=').
- A variable is automatically declared the first time we assign a value to it.
- Variable names:
  - must start with a letter or a underscore (_)
  - are case-sensitive
  - must consist of only letters, numbers or underscore
  - must not be a keyword (reserved words)

# First Python Program

- ```
  >>> print 'Hello World!'
  Hello World!
  ```
  `print` statement displays a value on the screen.
- ```
  >>> name = 'Joe'
  >>> print 'How are you %s?' %(name)
  How are you Joe?
  ```
- `print` automatically appends a newline while displaying the values

  ```
  >>> a, b = 5, 10
  >>> print a
  >>> print b
  5
  10
  ```
- you can avoid this by adding a ',' at the end of the `print` statement

  ```
  >>> print a,
  >>> print b
  5 10
  ```

# User input in Python

- you can get the variable input from user as follows:

```
>>> a = input("Enter a number: ")
Enter a number:  13
>>> print a
13
```

- `input()` can be used to get any type of objects from the user:

```
>>> a = input("Enter a list: ")
Enter a list:  [0,1,2,3]+[4,5,6]
>>> print a
[0,1,2,3,4,5,6]
```

- the `input()` function automatically converted the user input to appropriate type
- if you want to read the input as a string, use `raw_input()`:

```
>>> a = raw_input("Enter a number: ")
Enter a number:  13
>>> print a
'13'
```

# Python's Core Data Types

| | |
|---|---|
| Numbers | 1234, 3.1415, 999L, 3+4j, Decimal |
| Strings | 'spam', "guido's" |
| Lists | [1, [2, 'three'], 4] |
| Dictionaries | {'food': 'spam', 'taste': 'yum'} |
| Tuples | (1,'spam', 4, 'U') |
| Files | myfile = open('file1.txt', 'r') |
| Other types | Sets, Booleans, None |

# Python's Core Data Types

## Numbers

- integers, floating-point numbers, "long" integers, complex numbers
- supports the normal mathematical operations

```
>>> 123 + 222              # Integer addition
345
>>> 1.5 * 4                # Floating-point multiplication
6.0
>>> 2 ** 100               # 2 to the power 100
1267650600228229401496703205376L
>>> 5 % 3                  # Remainder on division of 5 by 3
2
>>> (-1+1j) + (2-3j)       # addition of complex numbers
1-2j
>>> 5 // 3                 # floor division
1
```

# Python's Core Data Types

## Numbers

- relation operations

```
>>> a, b = 2, 5
>>> a == b                  # Is a equal to b?
False
>>> a != b                  # Is a not equal to b?
True
>>> a < b                   # Is a less than b?
True
>>> a >= b                  # Is a greater than or equal to b?
False
```

# Python's Core Data Types

## Additional modules for Numbers

- `math` module - more advanced numeric tools

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871
```

- `random` module - random number generation and random selections

```
>>> import random
>>> random.random()
0.59268735266273953
>>> random.choice([1, 2, 3, 4])
1
```

# Python's Core Data Types

## Strings

- sequence of characters
- lots of built-in functions because of positional ordering

```
>>> S = 'spw2017'
>>> len(S)      # Length
7
>>> S[0]        # The first item in S, indexing starts from 0
's'
>>> S[1]        # The second item from the left
'p'
```

- strings are immutable, i.e., we cannot do the following:

```
>>> S[0] = 'a' # not allowed
```

# Python's Core Data Types

## Strings

- slicing - more general form of indexing

```
>>> S[0:3]          # everything in S from offset 0 upto offset 2
'spw'
>>> S[4:]           # Everything past the fourth (1:len(S))
'017'
>>> S[:3]           # Same as S[0:3]
'spw'
>>> S[:-1]          # Everything but the last
'spw201'
>>> S[:]            # All of S (0:len(S))
'spw2017'
```

- other operations

```
>>> S[0]+S[-2:]     # string concatenation
's17'
>>> S[0:2]*5        # string repetition
'spspspspsp'
```

# Python's Core Data Types

## Strings - built-in methods

- there are lots of built-in methods on strings like:

```
>>> S.find('2017') # lowest index where substring '2017' is found
3
>>> S.find('pa')   # substring not in the original string
-1
>>> S = 'How you doing?'
>>> S.split()      # split the string into list of strings
['How', 'you', 'doing?']
>>> S.split('o')   # split the string with delimiter 'o'
['H', 'w y', 'u d', 'ing?']
```

- you should also try these methods on your own: `count`, `isalpha`, `islower`, `strip`
- type the command `dir(S)` to see more methods for object `S`
- you can use `help(S.method_name)` (for e.g., `help(S.find)`) to know what the method does.

# Python's Core Data Types

## Lists

- positionally ordered collections of arbitrarily typed objects

```
>>> L = [1, [2,'three'], 4, 'five', 6.0]
>>> L = [[1,2], [2,3], 4, 5.0]
```

- lists are mutable unlike strings

```
>>> L[0] = 'a'          # valid assignment operation
```

- lists support all the sequence operations discussed for strings
- built-in methods:

```
>>> L.append('six')     # append new element to the list
>>> L
'a', [2,3], 4, 5.0, 'six']
>>> L.reverse()         # reverse the list
>>> L
['six', 5.0, 4, [2,3], 'a']
```

# Python's Core Data Types

## Dictionaries

- key-value mappings of objects stored by keys instead by relative position
- dictionaries are mutable like lists
- useful when we need to associate a set of values with keys

```
>>> D = {'a':19, 'b':8, 'c':11}
>>> D['d'] = 4          # create keys by assignment
>>> D['a'] += 1         # add 1 to value of 'a'
```

- built-in methods:

```
>>> D.keys()           # keys of the dictionary
['a','b','c','d']
>>> D.values()         # values of the dictionary
[20,8,11,4]
```

- you should try other built-in methods on your own

# Python's Core Data Types

## Tuples

- tuple is basically a list that cannot be changed
- tuples are sequences, like lists, but are immutable, like strings

```
>>> T = (1,2,3,4)          # a 4-item tuple
>>> T + (5,6)              # concatenation
(1,2,3,4,5,6)
>>> len(T)                 # length of tuple
6
>>> T[2] = 1               # not allowed; tuples are immutable
TypeError:  'tuple' object does not support item assignment
```

- Why do we need to use tuples?

# Python's Core Data Types

## Sets

- similar definition as mathematical sets
- 
```
>>> x = set('abcde')
>>> y = set('bdxyz')
>>> x
set(['a','b','c','d','e'])
>>> 'e' in x              # set membership
True
>>> x.difference(y)       # or 'x-y', set difference
set(['a','c','e'])
>>> x.union(y)            # set union
set(['a','c','b','e','d','y','x','z'])
>>> x&y                   # set intersection
set('b','d')
```

# Conditional Statements

## if-elif-else

- python `if` is similar to `if` statements in other languages like C / C++
- general format of if statements:

```
if <condition1>:          # if condition1 is True
    <statements1>         # associated block
elif <condition2>:        # optional elifs (elif - 'else if')
    <statements2>
else:                     # optional else
    <statements3>
```

- block of code associated with the first condition that evaluates to true is executed, or the else block if all conditions evaluate to false

# Conditional Statements

## Examples

- ```
  a = 17
  if a < 10:
      print 'a less than 10'
  elif a > 10:
      if a < 20:
          print 'a between 10 and 20'
      else:
          print 'a greater or equal to than 20'
  else:
      print 'a equal to 10'
  ```

- ```
  >>> a = 't' if 'wxyz' else 'f'      # nonempty is true
  >>> a
  't'
  >>> a = 't' if '' else 'f'          # empty is false
  >>> a
  'f'
  ```

# Control Flow - `for` and `while` Loops

- loops are statements that repeat an action over and over
- `while` statement provides a way to code general loops
- `for` statement is designed for stepping through the items in a sequence object, and running a block of code for each item
- general format of `while` statements:

```
while <condition>:          # loop condition
    <statements1>           # loop body
else:                       # optional else
    <statements2>           # run if didn't exit loop with break
```

- general format of `for` statements:

```
for <target> in <object>:   # Assign object items to target
    <statements1>           # Repeated loop body: use target
else:
    <statements2>           # If we didn't hit a 'break'
```

# Control Flow - for and while Loops

## Examples

- ```
  a=0; b=10
  while a < b:                    # One way to code counter loops
      print a,
      a += 1                      # Or, a = a+1
  ```

  Output: 0 1 2 3 4 5 6 7 8 9

- ```
  sum = 0
  prod = 1
  for x in [1, 2, 3, 4]:          # iterate over the list
      sum = sum + x
      prod = prod * x
  print sum, prod
  ```

  Output: 10 24

# Control Flow - for and while Loops

## break and continue

- break jumps out of the closest enclosing loop (past the entire loop statement)
- continue jumps to the top of the closest enclosing loop (to the loops header line)
- Loop else block: Runs if and only if the loop is exited normally (i.e., without hitting a break)
- Example:

```
x = 10
while x:
    x = x-1
    if x % 2 != 0:          # Odd? - skip print
        continue
    print x,
```

Output: 8 6 4 2 0

# Control Flow - for and while Loops

## break and continue

- Example:

```
y = 11
x = y / 2                       # For some y > 1
while x > 1:
    if y % x == 0:              # Remainder
        print y, 'has factor', x
        break                   # Skip else
    x = x-1
else:                           # Normal exit
    print y, 'is prime'
```

Output: 11 is prime

# For Further Reading I


Mark Lutz.
*Learning Python, Third Edition*.
OReilly Media, 2008.