AA 203
Optimal and Learning-based Control

Introduction to Python

Boris Ivanovic

Outline

- Python Language Basics
  - Variables and Basic Types
  - Containers (Lists, Tuples, Dictionaries)
  - Control Flow (If-Else Statements, For, While Loops)
  - Functions
  - Classes
  - Modules
- Some Examples

Introduction

- Python is an easy-to-use, general purpose programming language
- You will be using **Python 3** for your homework assignments and final projects for this class
- This session is intended to be a quick, but comprehensive introduction to the Python language basics and syntax

```
print(“Hello, World!”)
```

Variables

Python is a **dynamically-typed** language, meaning a variable is simply a name bound to a value. Variables are declared without explicitly specifying a type:

```
x = 2

y = 7.5

result = True

message = “Hello, World!”
```

Variables

However, objects and variables do have a type. These are the most basic and common types:

```
x = 2 #>> <type ‘int’>

y = 7.5 #>> <type ‘float’>

result = True #>> <type ‘bool’>

message = “Hello, World!” #>> <type ‘str’>
```

Can check variable/object type using: `type()`

Numeric Types and Math

Numeric types:

int and float

Basic Mathematical Operations:

Addition: 2 + 2 #>> 4

Subtraction: 8.0 – 3.0 #>> 5.0

Multiplication: 4.0 * 5 #>> 20.0

Division: 101 / 20 #>> 5.05 101 // 20 #>> 5

Exponentiation: 2**3 #>> 8

Modulo: 7 % 3 #>> 1

Numeric Types and Math

Important Point: An operation between two `ints` produces…

```
1 * 3 #>> 3 (int)                1 / 3 #>> 0.333… (float)
```

An operation involving a `float` produces a `float`:

```
1.0 / 3 #>> 0.333… (float)
1.0 / 3.0 #>> 0.333… (float)
```

Numeric type conversion follows order of operations in more complicated expressions:

```
(1 // 3) * 6 #>> 0 (int)
(1.0 // 3) * 6 #>> 0.0 (float)
```

Booleans

Boolean values in Python are `True` or `False`

```
result1 = True
result2 = False
```

Logical Operations:

```
not result1 #>> False
result1 and result2 #>> False
result1 or result2 #>> True
4 == 4 #>> True
4 != 3 #>> True
2 * 3 >= 8 #>> False
```

etc.

Strings

Declaring a string:

```
message = “Hello, World!”
```

Single quotes also work:

```
message = ‘Hello, World!’
```

String length:

```
len(message) #>> 13
```

Strings

Indexing Strings:

	0	1	2	3	4	5	6	7	8	9	10	11	12	
“	H	e	l	l	o	,		W	o	r	l	d	!	”
	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

Note: Python indexing begins with 0!

Strings

	0	1	2	3	4	5	6	7	8	9	10	11	12	
“	H	e	l	l	o	,		W	o	r	l	d	!	”
	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

String Indexing/Slicing:

```
message[0] #>> ‘H’
message[–1] #>> ‘!’
message[1:5] #>> ‘ello’
message[:5] #>> ‘Hello’
message[2:] #>> ‘llo, World!’
message[2:10:2] #>> ‘lo o’
```

Containers

Lists:
Represent ordered, **mutable** collections of objects of potentially different types. We can add or remove elements from a list at will.

Examples:

```
numbers = [1, 2, 3, 4, 5]
colors = [“red”, “green”, “blue”]
numbers_and_colors = [3, “red”, 6.7, “blue”]
```

Empty list:

```
empty_list = []
empty_list = list()
```

Concatenation:

```
numbers + colors
#>> [1, 2, 3, 4, 5, ‘red’, ‘green', ‘blue’]
```

Containers

Lists:
Indexing/slicing is performed just as shown for strings, however now the list elements may be modified:

```
colors = [“red”, “green”, “blue”]
colors[1] = 10
#>> colors = [“red”, 10, “blue”]
```

Relevant Functions and Methods:
`len()`, *list*`.append()`, *list*`.pop()`, *list*`.remove()`, *list*`.sort()`, …

More details on lists:
https://docs.python.org/3/tutorial/introduction.html#lists
https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Containers

Tuples:
Ordered **immutable** (i.e. fixed) collections of objects of potentially different types.

Examples:

```
numbers = (1, 2, 3, 4, 5)
colors = (“red”, “green”, “blue”)
numbers_and_colors = (3, “red”, 6.7, “blue”)
```

Empty tuple:

```
empty_list = ()
empty_list = tuple()
```

Concatenation:

```
numbers + colors
#>> (1, 2, 3, 4, 5, ‘red’, ‘green', ‘blue’)
```

Containers

Tuples:
Indexing/slicing is performed as before, but now the elements can **not** be modified:

```
colors = (“red”, “green”, “blue”)
colors[1] = 10
#>> TypeError: ‘tuple’ object does not support item assignment
```

Why use tuples as opposed to lists?
Tuples are slightly faster and smaller than lists. As such, they are useful for constant sets of values of fixed size.

Containers

Dictionaries:
Used to store **unordered*** key:value pairs. Dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys.

Example:

```
capitals = {"California":"Sacramento",
            "Massachusetts":"Boston",
            "New York":"Albany"}

print capitals["New York”] #>> Albany
```

***They are actually order-preserving in Python 3, but it’s not the best idea to rely on that.**

Control Flow

Conditionals:

```
if <condition 1>:
    <perform action 1>
elif <condition 2>:
    <perform action 2>
else:
     <perform action 3>
```

Note: Statements are grouped by indentation.

4 spaces or 1 tab.

Control Flow

Conditionals:

```
number = 10
if number > 0:
    print(“number is positive.”)
elif number < 0:
    print(“number is negative.”)
else:
    print(“number is zero.”)
```

Control Flow

For Loops:

List, Tuple, String, etc.

```
for <element> in <iterable>:
    <do something with element>
```

Control Flow

For Loops:

```
number_list = [10, 5, 6, 3, 7]
for number in number_list:
    square = number**2
    print(square)

#>> 100
#>> 25
#>> 36
#>> 9
#>> 49
```

Control Flow

For Loops:

`range()` is used to create lists of numbers

```
range(5) #>> [0, 1, 2, 3, 4]

range(1, 10) #>> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Additional detail:
https://docs.python.org/3/library/functions.html#func-range

Control Flow

For Loops:

```
number_list = [1, 2, 3, 4, 5]
for i in range(len(number_list)):
    number_list[i] = number_list[i]**2
print(number_list)

#>> [1, 4, 9, 16, 25]
```

Control Flow

For Loops (List Comprehension):

```
number_list = [1, 2, 3, 4, 5]
squares = [number**2 for number in number_list]

#>> [1, 4, 9, 16, 25]
```

Control Flow

While Loops:

```
while <condition>:
    <do something>
```

Functions

```
def <function name>(<arguments>):
    <function body>
  <return statement> (optional)
```

Example:

```
def circle_area(radius):
    pi = 3.14
    area = pi * radius**2
    return area

r = 2
print(circle_area(r))

#>> 12.56
```

Classes

```
class Cube:
    def __init__(self, edge_length):
        self.edge_length = edge_length

    def face_area(self):
        return self.edge_length**2

    def volume(self):
        return self.face_area() * self.edge_length

my_cube = Cube(3)
print(my_cube.edge_length) #>> 3
print(my_cube.volume()) #>> 27
```

Classes

Function called automatically when class is called to create object.

```
class Cube:
    def __init__(self, edge_length):
        self.edge_length = edge_length

    def face_area(self):
        return self.edge_length**2

    def volume(self):
        return self.face_area() * self.edge_length

my_cube = Cube(3)
print(my_cube.edge_length) #>> 3
print(my_cube.volume()) #>> 27
```

Classes

Reference to the class instance itself. Must be first parameter of any function in the class.

```
class Cube:
    def __init__(self, edge_length):
        self.edge_length = edge_length

    def face_area(self):
        return self.edge_length**2

    def volume(self):
        return self.face_area() * self.edge_length

my_cube = Cube(3)
print(my_cube.edge_length) #>> 3
print(my_cube.volume()) #>> 27
```

Note use of self.

Modules

Python modules (i.e. libraries) can be imported in a few ways. Import statements are usually included at the beginning of a script.

```
import math
print(math.pi) #>> 3.14159265359

import math as m
print(m.exp(1)) #>> 2.71828182846

from math import pi, sin
print(sin(pi/2)) #>> 1.0
```

For more about the math module: https://docs.python.org/3/library/math.html

Modules

Additional modules you’ll be seeing throughout the assignments include:

- numpy: http://www.numpy.org/
- scipy: https://www.scipy.org/
- matplotlib: https://matplotlib.org/

Modules

Python modules can also be used to import functions and classes from other scripts you have written:

Does **not** include .py extension.

```
import <filename>

<filename>.<function/class>()
```

Can also import other scripts using:

```
import … as …
from … import …
```

Additional Resources

Many more tutorials and plenty of documentation can be found online, some of which are:

- https://docs.python.org/3/tutorial/index.html
- http://cs231n.github.io/python-numpy-tutorial/