Component-based Software Development

Python Programming – An Intro

Dr. Vinod Dubey
SWE 645
George Mason University

ACKNOWLEDGEMENT

Information in this lecture are adapted from

https://www.python.org/

https://www.w3schools.com

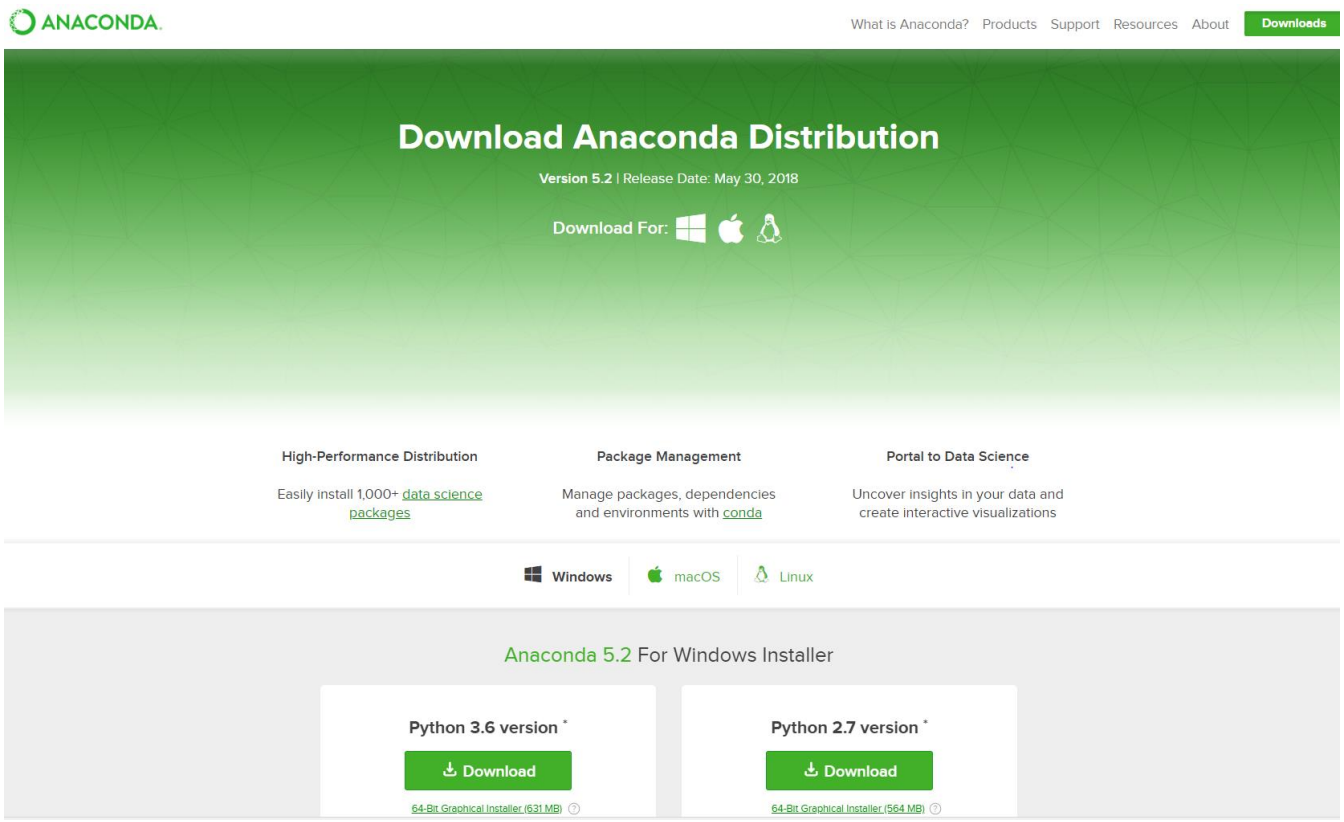
&

http://google.com

Table of Contents

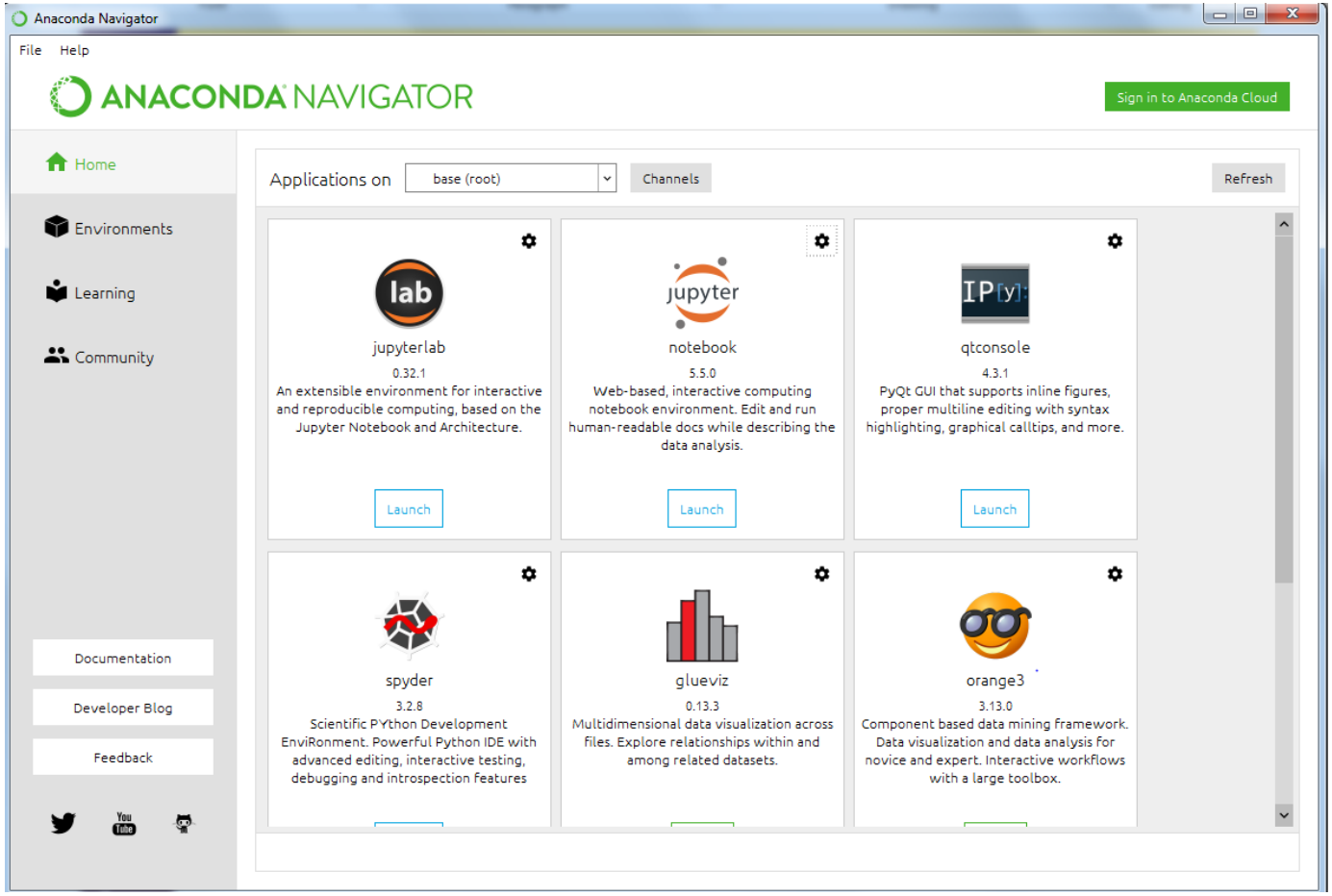
Anaconda Installation

- **Using Anaconda distribution**
  - https://www.anaconda.com/download/



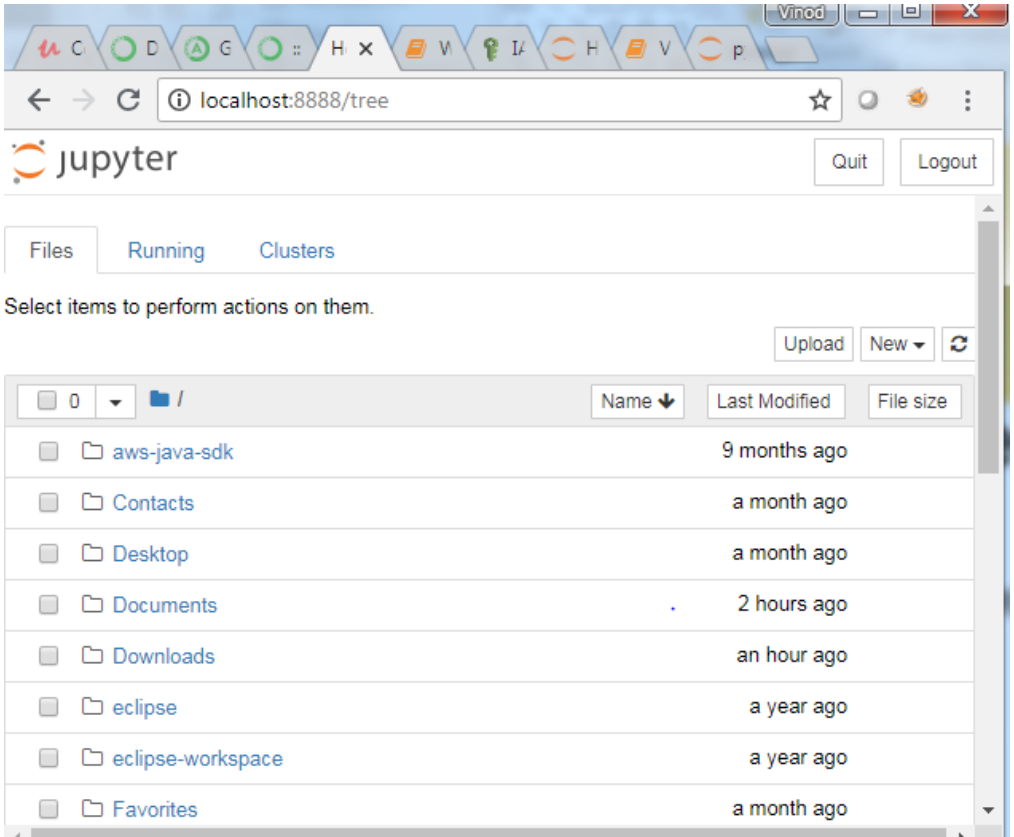
Anaconda Navigator

- **Once Anaconda is installed, open Anaconda Navigator from Windows start menu**



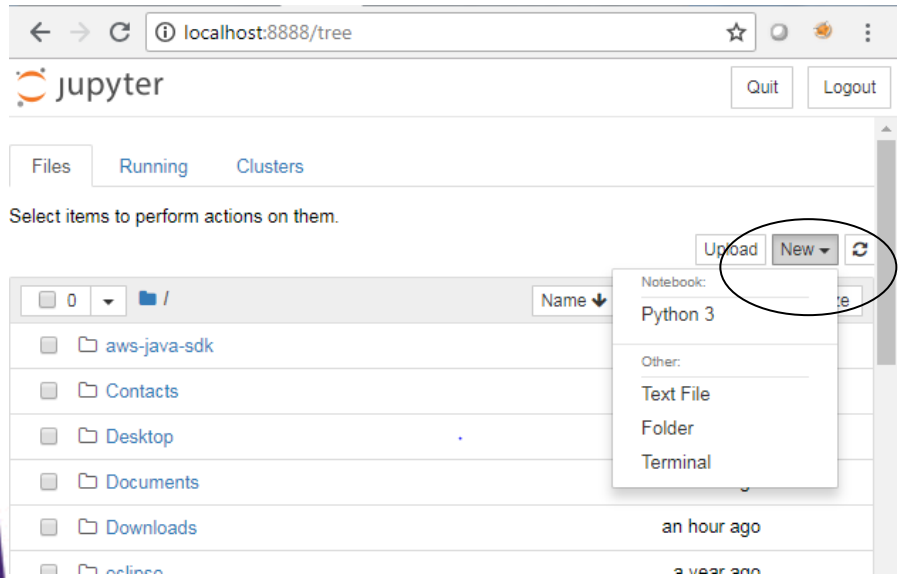
Jupyter Notebook Console

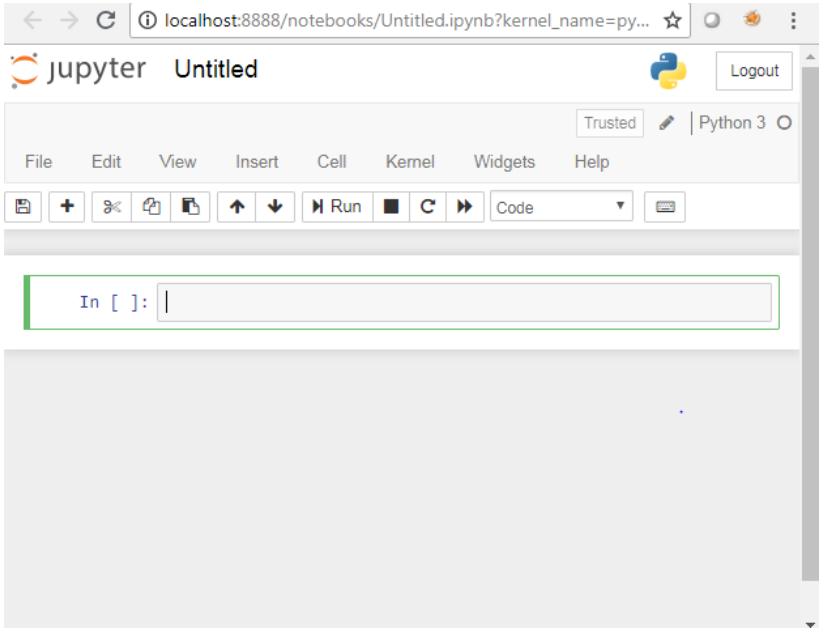
- **Open Jupyter Notebook Console by clicking the Jupyter icon in Anaconda Navigator**
  - The navigator allows you to navigate your directories/files on your desktop



Jupyter Notebook

- **Open Jupyter Notebook by selecting the Python3 icon in the dropdown labeled New**





Useful links

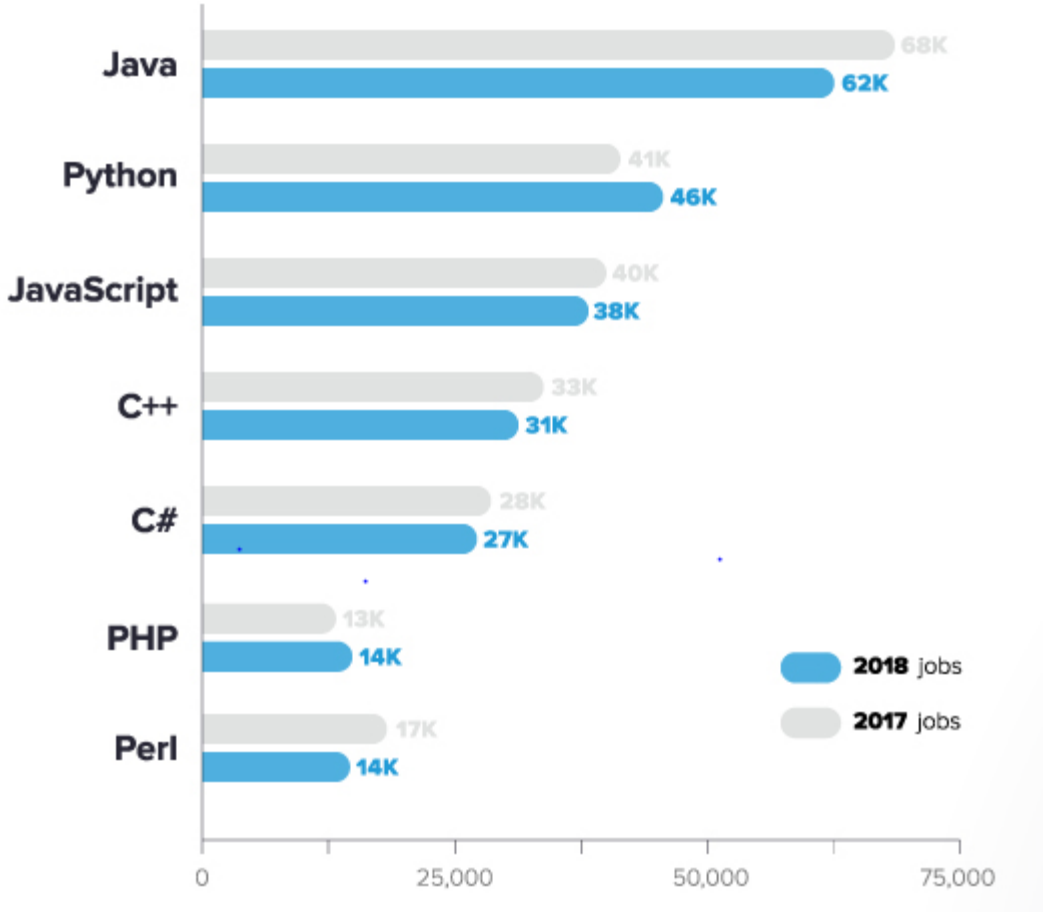
- **Useful links to learn python**
  - https://www.w3schools.com/python/
  - http://jupyter.org/try

Pyathon - Introduction

- **Python is an interpreted, high level, general purpose programming language**
  - A dynamically typed language that features automatic memory management, and emphasizes code readability by using significant whitespaces
- **A language of choice in data science community**
  - Popular libraries: NumPy, Pandas, Matplotlib, scipy
  - Created by Guido van Rossum

Python Popularity

- **The 7 Most In-Demand Programming Languages of 2018**



Sourse:
https://www.codingdojo.com/blog/7-most-in-demand-programming-languages-of-2018

At a glance: Java vs. Pyathon

- **No type declarations of variables in Python**
  - Python is a dynamically typed language

	Java	Python

Simple arithmetic	`int b = 10;`
	`int a = b + 1;`
	`a = 20 * 5;`
	`a += b;`
	`a *= b;`
		`b = 10`
		`a = b + 1`
		`a = 20 * 5`
		`a += b`
		`a *= b`

Boolean arithmetic	`a > b && c == d`
	`a > b || c < d`
	`!x`
	`true`
	`false`
		`a > b and c == d`
		`a > b or c < d`
		`not x`
		`True`
		`False`

Conditional	`if ( <boolean exp> ){`
	`  ...`
	`} else if ( <boolean exp> ){`
	`  ...`
	`} else {`
	`  ...`
	`}`
		`if <boolean exp>:`
		`  ...`
		`elif <boolean exp>:`
		`  ...`
		`else:`
		`  ...`

For loop	`for (int i = 0; i < n; i++){`
	`  ...`
	`  System.out.println(i);`
	`}`
		`for i in range(n):`
		`  ...`
		`  print i`

Foreach	`for (int x: arrayA){`
	`  ...`
	`  System.out.println(x);`
	`}`
		`for x in array_a:`
		`  ...`
		`  print x;`

At a glance: Java vs. Pyathon

- **Notice the use of *semicolon* :**
  - Syntax requires **:** after if, for, while, function def, etc.
- **The use of indentation**
  - Indentation, instead of {}, to specify blocks of code

	Java	Python

While loop	`while ( <boolean exp> ){`
	`  ...`
	`}`
		`while <boolean exp>:`
		`  ...`

Print	`System.out.println("Hello");` `System.out.print("Hello");`	`print "Hello"` `print "Hello",`

Array	`int[] a = new int[3];`
	`a[0] = 1;`
	`a[1] = 2;`
	`a[2] = 3;`
		`a = [1, 2, 3]`

Function/Method	`int add(int a, int b){`
	`  return a + b;`
	`}`
		`def add(a, b) :`
		`  return a + b`

Try-Catch	`try{`
	`  ...`
	`} catch(MyException e){`
	`  throw new Exception("Error");`
	`}`
		`try:`
		`  ...`
		`except MyException as e:`
		`  raise Exception("Error")`

Python Types

- **Python list of built-in data types**
  - –None
  - –Booleans
  - –Numbers (i.e., int, float, long, complex)
  - –Strings
  - –Lists, Tuples, Sets
  - –Dictionaries
- **Data science libraries add new data types to Python**
  - ndarray (NumPy)
  - Series (Pandas) and
  - DataFrame (Pandas)

Python Variables

- **Created when it's assigned a value**
  - Don't need to be declared of any particular type
  - Can even change type after they have been set
- **Python variable names:**
  - Start with a letter or the underscore character
    - Cannot start with a number
  - Only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Are case-sensitive

Python Variables

- **Example declaring variables in Python**

```
a = 8                    # => variable a refers to 8
b = 9                    # => variable b refers to 9
c = "Hello World"        # => variable c refers to "Hello String"
b = "Hello There"        # => Reassign different values a variable (even of different types)
```

Food for thought!

```
How to swap the the values held by b and c?
```

Python Variables–More examples

- **Example declaring and using variables in Python**

```
a = 8                       # => variable a refers to 8
b = 9.5                     # => variable b refers to 9.5

print (a+b)                 # => prints 17.5

c = "CLoud "                # => variable c refers to "Cloud "
d = 9                       # => variable d refers to 9

mylist = [3,5,7,8,9]        # => variable mylist refers to a list

str1 = "Hello There"        # => variable str1 refers to "Hello There"
str2 = "Hello World"        # => variable str1 refers to "Hello World"

print(str1 + ' ' + str2) # => Hello There Hello World // string concatnation

print( c + d)          # =>  TypeError: must be str, not int;
                       # => Combining string with an numeric value (e.g., int) not allowed

print (c + str(d))          # => prints CLoud 9
```

String in Python: str type

- **Sequence of characters in 'single' or "double quotes" – string literals**
  - 'Hello' is same as "Hello"
    - In computer science, a literal is a notation to represent a fixed value in source code
    - Python does not have a character data type, a single character is simply a string with a length of 1.
  - Zero-based indexing or slicing to access parts of the string

```
mystr = 'Hello'

print( mystr )              # =>    Hello

print( mystr[1] )           # =>    e

print( len(mystr) )         # =>    5

print( mystr + ' there' )   # =>    Hello there
```

str Methods

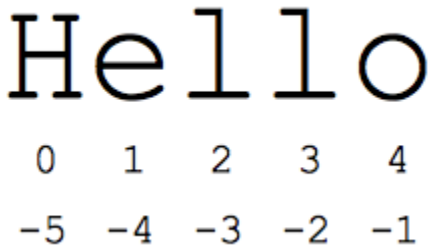
- **Python has a built-in string class "str" with many handy features**
- **Let mystr refers to a str literal**
  - mystr.lower(), mystr.upper()
    - returns the lowercase or uppercase version of the string
  - mystr.isalpha() / mystr.isdigit()
    - tests if all the string chars are in the various character classes
  - mystr.startswith('other'), mystr.endswith('other')
    - tests if the string starts or ends with the given other string
  - mystr.find('other')
    - searches for the given other string within mystr
    - returns the first index where it begins or -1 if not found

str Methods

- **Other handy features**
  - mystr.replace('old', 'new')
    - returns a string where all occurrences of 'old' have been replaced by 'new'
  - mystr.split('delim')
    - returns a list of substrings separated by the given delimiter
    - mystr.split() (with no arguments) splits on all whitespace chars
  - mystr.join(list)
    - opposite of split() – joins the elements in the given list together using the mystr as the delimiter
    - e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc
  - mystr.strip()
    - returns a string with whitespace removed from the start and end

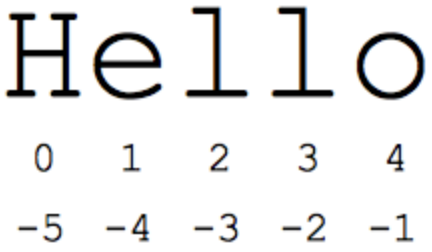
String Slicing

- The "slice" syntax to grab sub-parts of sequences
  - typically strings and lists.
- Syntax: mystr[start:end:step]
  - Elements beginning at start and extending up to but not including end. By default, step=1
  - mystr[1:4] is 'ell'
  - mystr[1:] is 'ello'
    - omitting either index defaults to the start or end of the string
  - mystr[:] is 'Hello'
    - omitting both always gives us a copy of the whole thing



Reverse Index

- **The zero-based index numbers**
  - Provides easy access to chars near the start of the string
- **Python also uses revers index (i.e., negative numbers for index)**
  - To give easy access to the chars at the end of the string
  - Negative index numbers count back from the end of the string:



  - s[-1] is the last char 'o'
  - s[-2] is 'l' the next-to-last char, and so on

Strings Immutability

- **Strings are immutable**
  - Once assigned a value to a string, it can't be mutated or changed.
- **For example**
  - mystring = 'hello'
  - The following is NOT allowed:
  - mystring[0] = 'z' // Error: item assignment not supported by str object

Strings Concatenation

- **Assume**
  - mystr = ‘Hello’
- **Concatnation examples:**
  - print(mystr + mystr) # => HelloHello
  - print(“Hi “ + “ John”) # => Hi John

```
mystr = 'Hello'

print( mystr + mystr)          # => HelloHello

print ('Hello ' + 'John')      # => Hello John
```

String Print formatting

- **Using .format method**
  - Python inserts format variables in {} in the order specified

```
name = 'John'
color = 'blue'

print('The favorite color of {} is {}'.format(name, color))
```

```
The favorite color of John is blue
```

- **Using f string literal**
  - Must precede your string with f
  - Put variables inside {}

```
print(f"The favorite color of {name} is {color}")
```

```
The favorite color of John is blue
```

Python Operators

- **Python supports the following operators**
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators

Python Operators

Arithmetic Operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division

Comparison Operators

Operator	Name
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Python Operators

- **Logical operators**
  - Used to combine conditional statements
  - Python uses English words and, or, not rather than symbols (&&, ||, etc.)

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Operators

- **Membership operators**
  - Used to test if a sequence is present in an object

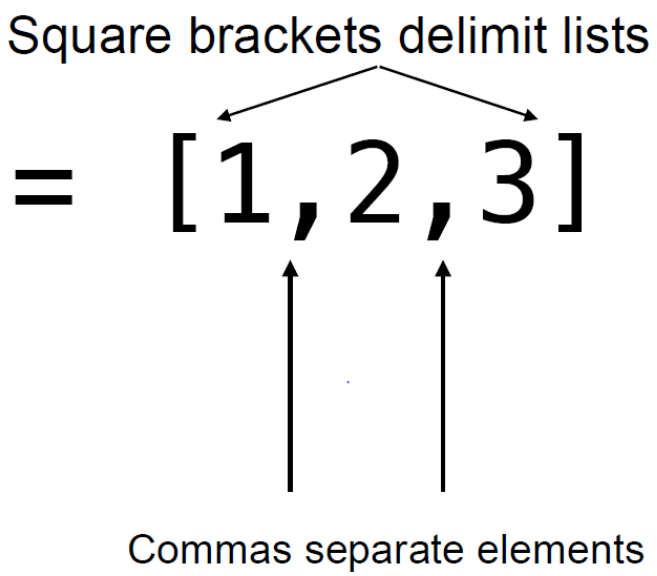
Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Collections – container types

- **Python includes several built-in container types**
  - list
  - tuple
  - set
  - dict – Dictionaries

Lists

- Lists are ordered sequences with comma-separated values (items) between square brackets
- The Python equivalent of an array, but is resizable and may contain elements of different types
- Lists are mutable



Lists

- The most important list method is append(), which adds a new element to the end of a list. For example:
  - animal_types = ['bird', 'giraffe', 'monkey']
  - animal_types.append('turkey')
- After evaluating both lines, animal_types contains four elements : ['bird', 'giraffe', 'monkey', 'turkey'].
- Another important list method is .pop() which removes an element from a list.

Common List Methods

List Method	Description
list.append(elem)	Adds a single element to the end of the list.
list.insert(index, elem)	Inserts the element at the given index, shifting elements to the right.
list.index(elem)	Searches for the given element from the start of the list and returns its index.
list.remove(elem)	Searches for the first instance of the given element and removes it
list.sort()	Sorts the list in place (does not return it).
list.reverse()	Reverses the list in place (does not return it)
list.pop(index)	Removes and returns the element at the given index.

Common List Methods

List Methods, Continued

- **Unlike string methods, the list methods alter the original list.**
  - The methods of most containers (excluding strings and tuples) alter the same object for efficiency's sake.
  - On the other hand, primitive data types— Booleans, numbers, strings — are immutable
    - An immutable object never changes its value.
    - Instead, the interpreter creates a new object and directs the name to it.
    - For example, evaluating 312 + 457 actually creates a new int object, 769.

Lists

- **List Examples**

```
# Create a new list
empty = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]

# Lists can contain elements of different types
mixed = [4, 5, "seconds"]

# Append elements to the end of a list
numbers.append(7)   # numbers == [2, 3, 5, 7]
numbers.append(11)  # numbers == [2, 3, 5, 7, 11]
```

Lists

- **More List Examples**

```
# Access elements at a particular index
numbers[0]  # => 2
numbers[-1]  # => 11

# You can also slice lists - the same rules apply
letters[:3]  # => ['a', 'b', 'c']
numbers[1:-1]  # => [3, 5, 7]

# Lists really can contain anything - even other lists!
x = [letters, numbers]
x  # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
x[0]  # => ['a', 'b', 'c', 'd']
x[0][1]  # => 'b'
x[1][2:]  # => [5, 7, 11]
```

Lists

- **Nested List – a list that contains other list(s)**
  - For example,

```
list1 = [0,1,2]
list2 = [3,4,5]
list3 = [6,7,8]
mega_list = [list1, list2, list3]
print(mega_list) # => [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

```
len(mega_list)     # => 3
mega_list[2]       # => [6, 7, 8]
mega_list[2][2]    # => 8
```

Lists

- **Lists are ordered.**
  - Their elements have a particular order that will never change.
- **Lists are heterogeneous.**
  - Different data types can be stored for each element in the list. For example, ['cat', 10, 0.4].
- **Lists are mutable.**
  - When you alter a list, you don't create a new object — the original object is just modified.
- **Typically, the name of a list should be plural.**
  - For example, cars, animal_names, and cities.
  - This immediately indicates that the name refers to a container.

Literals

- **In general, a literal is a text representation of a particular data type.**
  - For example, 'I live in a city.' is a string literal.
  - However, I live in a city does not represent a string

Tuples

- A tuple is an ordered collection of Python objects separated by commas inside a pair of parenthesis

```
mytuple = ('apple', 'orange', 'mango')

print(mytuple[1])          # => orange

print(mytuple[2])          # => mango
```

Tuples

- **Tuples are similar to lists in that they are containers of objects.**
- **However, there are two main differences**
  - They are defined using parentheses instead of brackets.
  - They are immutable — you cannot alter a particular tuple object once it is created.
  - So, most methods such as append() and pop() do not exist.
- **You can define a new tuple like:**

```
month = (1, 'January', 31)          # (month number, month name, num days)
```

Tuples

- **A tuple is similar to a List in terms of indexing, slicing, nested objects, but a tuple is immutable**
  - Once elements are assigned inside a tuple, it cannot be reassigned

```
t = (2, 50, 70, 8, 23.3, 'John', 50, (1,2,3), 61.9)

print(t[:4])    # slicing =>  (2, 50, 70, 8)

print(t[-1])    # last element   => 61.9

print(t[5])     # 'John'

t[5] = 'Jane'   #Error: item reassignment not supported
```

Lists vs. Tuples

- **Use lists when you intend to mutate (change) the elements within it**
- **Use tuples when you do not intend to mutate the elements within it**
- **Elements within both lists and tuples retain their order**

Lists vs. Tuples

- **Why use a tuple instead of a list?**
  - Tuples are generally used as containers of protected constants - for example, the acceleration due to gravity, or a connection URL to a server.
  - Because they are immutable, they can't accidentally be overwritten.
- **Because tuples have fixed sizes (determined when they are assigned their initial values), they are more memory efficient than a list,**
  - which needs additional memory allocated to it.
- **If you aren't going to be adding elements to your object, a tuple may be a better choice.**

Sets

- **A set is an unordered collection of unique elements**
  - In Python sets are written with curly brackets.

```
s = {'apple', 'orange', 'banana', 'cherry', 100, 20, 50}
```

- **Can create a set from a list by passing the list to set()**

```
mylist = [2, 5, 6, 4, 4, 200, 2, 1, 100, 100]

print(mylist)  # => [2, 5, 6, 4, 4, 200, 2, 1, 100, 100]

myset = set(mylist)
print(myset)   # => {1, 2, 4, 5, 6, 100, 200}
```

Sets

- **Can use the set() constructor to make a set**
  - add() method to add an item
  - remove() method to remove an item from the set
  - len() function returns the size of the set
- **The set list is unordered, so the items can appear in a random order**

```
s = set()
s.add(10)
s.add(20)
s.add(20)
s.add(30)
s.add(5)

print(s)      # => {10, 20, 5, 30}
```

```
thisset = set(("apple", "banana", "cherry"))
print(len(thisset))       # => 3
```

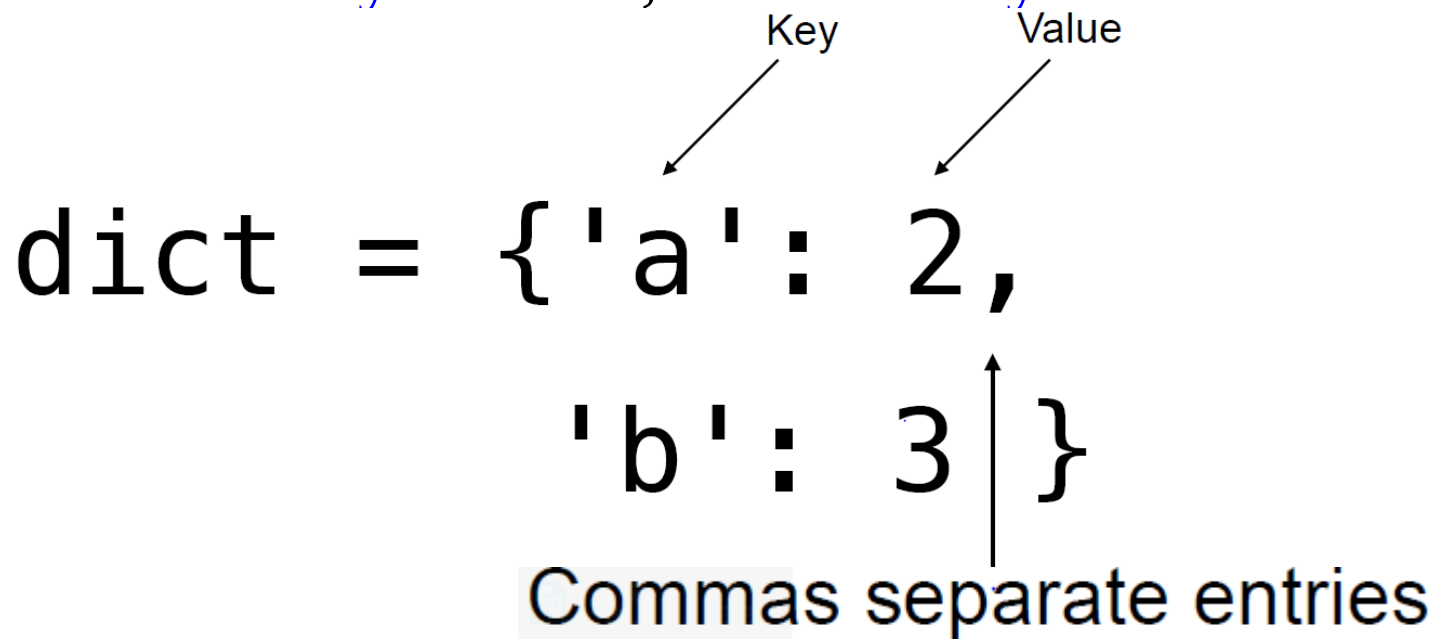
Sets

- **Food for thought!**
  - What would be the result of evaluating the following code? 7, 10, 12, or 14?

```
len({3, 2, 9, 5, 3, 6, 2}) + len([3, 2, 9, 5, 3, 6, 2])
```

Dictionary

- **A dictionary represents or stores (key, value) pairs**
  - similar to a Map or HashMap in Java
- **An unordered collection of mapping, which is changeable**
  - Written with curly brackets, and have keys and values



Dictionary

- **As with sets, a dictionary is defined using curly braces.**
  - However, each element of a dictionary consists of a key, followed by a colon, followed by a value.
- **For example:**

book_authors = {'Moby Dick': 'Herman Melville', 'The Lorax': 'Dr. Seuss', 'The Hobbit': 'J.R.R. Tolkien'}

  - Notice this is a container of key-value pairs.

So, len(book_authors) is 3, as there are three key-value pairs.

- **Two useful dictionary methods are:**
  - Keys(), e.g. book_authors.keys()
  - Values(), e.g., book_authors.values()

Dictionary Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Dictionary

- **Creating and accessing dictionaries**

```
d = {'username': 'Jane', 'password': 'pass123', 'id': 101}

print(d['username'])      # => Jane

print(d['password'])      # => pass123
```

Dictionary

- **Updating dictionaries**
  - Keys must be unique
  - Assigning to an existing key replaces its value
  - Dictionaries are unordered
    - New entry might appear anywhere in the output

```
d['password'] = 'passXYZ'

print(d)      # => {'username': 'Jane', 'password': 'passXYZ', 'id': 101}

d['department'] = 'CS'

print(d) # => {'username': 'Jane', 'password': 'passXYZ', 'id': 101, 'department': 'CS'}
```

Dictionary

- **Removing dictionary entries:**
  - d.pop('key') removes the corresponding entry from the dictionary d
  - d.clear() removes all elements from the dictionary d

```
d['dept'] = 'CS'

print(d) # => {'username': 'Jane', 'password': 'passXYZ', 'id': 101, 'department': 'CS', 'dept': 'CS'}

d.pop("department")

print(d)   # =>  {'username': 'Jane', 'password': 'passXYZ', 'id': 101, 'dept': 'CS'}

d.clear()

print(d)    #  =>   {}
```

Dictionary

- **Access methods: keys(), values(), items()**
  - List of current keys

```
d = {'username':'Jane', 'password': 'pass123', 'id':101}
```

```
d.keys()        #  => ['username', 'password', 'id']
```

  - List of current values

```
d.values()      #  => ['Jane', 'pass123', 101]
```

  - List of item tuples

```
d.items() # => [('username', 'Jane'), ('password', 'pass123'), ('id', 101)]
```

Dictionary

- **Values in a dictionary can be a list as well**

```
people = {'John':[1,2,3], 'Kelly':[10,20,30]}

print(people['John'])       # => [1, 2, 3]

print(people['Kelly'])      # => [10, 20, 30]
```

- **Nested dictionary**

```
people = {'John':{'salary':100, 'age':22}, 'Jane':{'salary':150, 'age': 30}}

print(people['Jane'])           #  =>  {'salary': 150, 'age': 30}

print(people['Jane']['age'])    #  =>  30
```

Booleans

- **Boolean variable can be either True or False**
  - Notice the uppercase T and F in True/False
  - Lowercase true and false are not correct!

```
x = True
y = False

print(f"A boolean variable can be either {x} or {y}")

# => A boolean variable can be either True or False

z = true
print(f"Not allowed to assign lowercase true to {z}")

#Error: A boolean variable can be either True or False
```

Boolean Expressions

- **Comparison Operators**
  - Comparison operators are used to compare two values

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Boolean Expressions

- **Logical Operators**
  - Logical operators are used to combine conditional statements

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Control Flow

- **Allows to execute certain code only when a particular condition is true**
  - if statements
  - while loop
  - for loop
- **Control flow syntax uses**
  - colon (:) and
  - indentation

Control Flow

- **If statements**

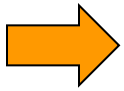
```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
print "This is outside the 'if'."
```

- **Note:**
  - Use of indentation for blocks
  - Colon (:) after Boolean expressions

Control Flow

- **While loop -** executes a block of code as long as a condition is true

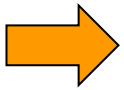
```
num = 1

while (num < 5 ):
    print(num)
    num = num + 1
```



```
1
2
3
4
```

With continue

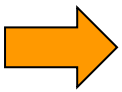
```
num = 0
while num < 10:
    num += 1
    if ( num%2 == 0):
        continue
    print(num)
```



```
1
3
5
7
9
```

With break

```
num = 1
while num < 10:
    print(num)
    if num == 5:
        break
    num += 1
```



```
1
2
3
4
5
```

Control Flow

- **For loop -** is used for **iterating** over a collection
  - The **collection** could be a list, tuple, dictionary, or string
  - Executes statement(s), once for each item in a collection

```
#If the collection is a list or tuple,
#then loop through each item
for <item> in <collection>:
    <statements>

#If the collection is string, then loop through each char
for myChar in "Hello World":
    print(myChar)

#You loop through keys of a dictionary
for key in my_dictionary:
    print(key)
```

Control Flow

- **For loop –** with **list**

```
mylist = [1, 2, 3, 4, 5]

for num in mylist:
    print (num**2)     #
```

```
1
4
9
16
25
```

- **For loop –** with **string**

```
mystring = 'hello'

for char in mystring:
    print(char)
```

```
h
e
l
l
o
```

- **For loop –** with **dictionary**

```
mydictionary = {'one': 1, 'two': 2, 'three': 3}

for key in mydictionary:
    print(key)


for key in mydictionary:
    print(key + " = " + str(mydictionary[key]))
```

```
one
two
three

one = 1
two = 2
three = 3
```

Comprehension in Python

- **Comprehension is used to create a new built-in data structure like a list, a dictionary or a set from an existing data structure**
  - generally to transform one type of data into another

- **Python has three types of comprehension**
  - List comprehension
  - Set comprehension
  - Dictionary comprehension

Comprehension in Python

- **Comprehension usually involves the following steps:**
  - Begin with a new empty data structure to create, for example, a list, set or a dictionary
  - Loop through an existing data structure and process each item in the iterator
  - Populate the new data structure with the processed data

List Comprehension

- **Syntax**

```
new_list = [expression for each_item in existing_data_structure]
```

  - The outer [] brackets indicates the creation of a new list for each item in the iterator.
  - The expression can be any valid python expression resulting into a value to include in the newly created iterator.

List Comprehension

An example to compute square numbers

- **Syntax**

```
new_list = [expression for each_item in existing_data_structure]
```

- **Without list comprehension**

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print squares
```

```
[0, 1, 4, 9, 16]
```

- **Using list comprehension**
  - Same effect with simpler code

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print squares
```

```
[0, 1, 4, 9, 16]
```

  - List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print even_squares
```

```
[0, 4, 16]
```

Set Comprehension

- Creating a set from an existing data structure
- Similar to list comprehension, but you substitute [] with {} in set comprehension.
- The opening and closing curly braces '{}' means to start with an empty set and create it from an existing data structure.
- Syntax

```
{ expression for each_item in existing_data_structure }
```

Set Comprehension

- **Syntax**

```
{ expression for each_item in existing_data_structure }
```

- **Example**

```
# Get a set of even numbers from a list using set comprehension
numbers = [1, 24, 35, 12, 67, 2, 24, 81, 2, 45, 89, 12]
even_numbers = {each_number for each_number in numbers if each_number%2 == 0}
print(even_numbers)
```

```
{24, 2, 12}
```

```
from math import sqrt
print ({int(sqrt(x)) for x in range(10)})
```

```
{0, 1, 2, 3}
```

Dictionary Comprehension

- **Used for creating a new dictionary from an existing data structure**
  - The existing data structure doesn't need to be a dictionary
  - It can as well be a list or a set or a tuple.
- **Similar to list comprehensions, but allows to easily construct dictionaries**
- **Syntax**

```
{ key:value for key,value in existing_data_structure }
```

Dictionary Comprehension – Examples

- **Creating a dictionary from a list**

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print even_num_to_square
```

```
{0: 0, 2: 4, 4: 16}
```

- **Creating dictionary from another dictionary**

```
students = {"dennis": 23, "david": 21, "mary": 9, "daniel": 25, "darius": 17, "jim": 10}

students_with_upper_case_names = {key.upper(): value for key, value in students.items()}

print(students_with_upper_case_names)
```

```
{'DENNIS': 23, 'DAVID': 21, 'MARY': 9, 'DANIEL': 25, 'DARIUS': 17, 'JIM': 10}
```

Functions in Python

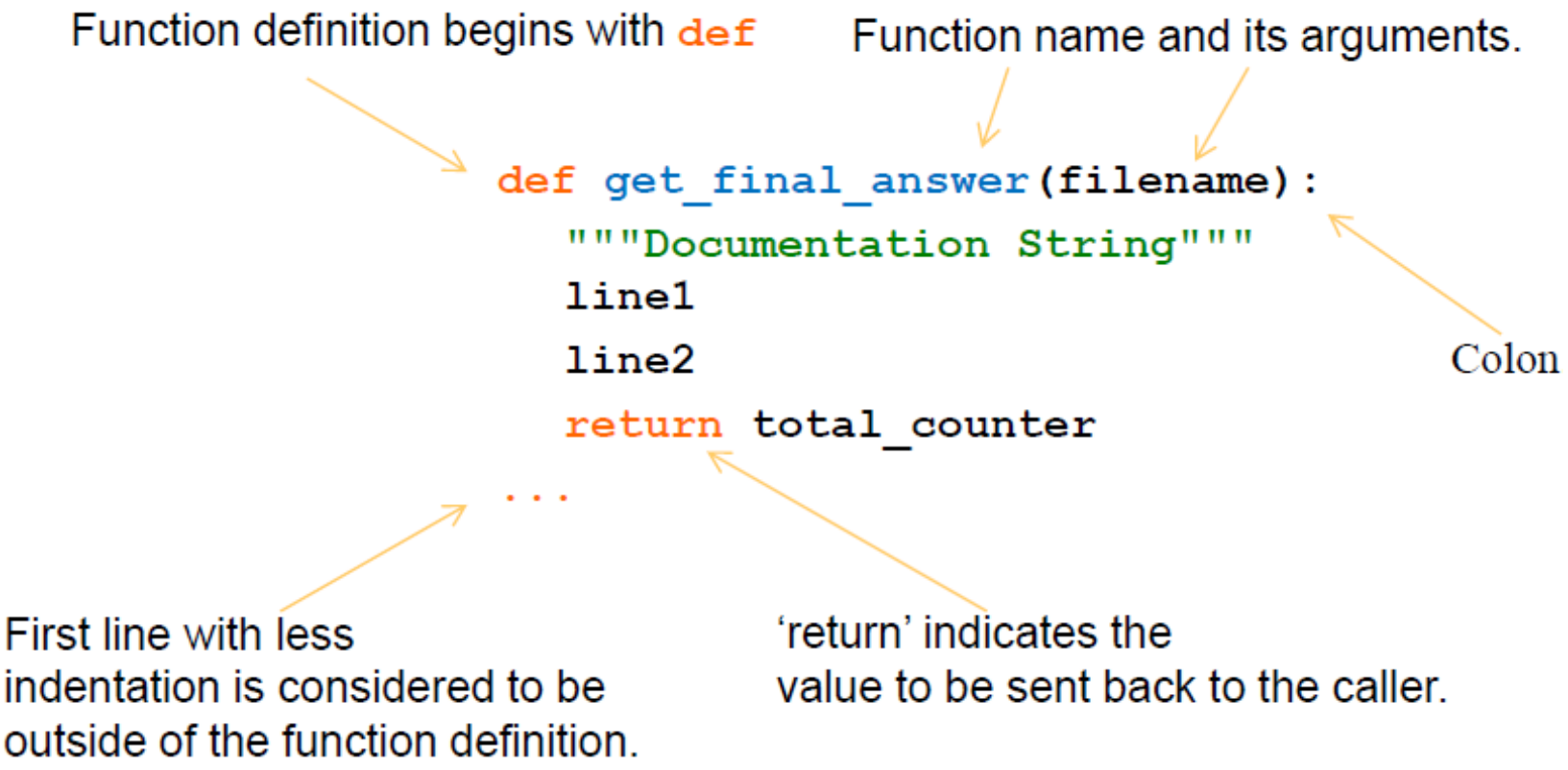
- **A function defines block of code**
  - Executes when the function is called
  - Promotes **reusability** of the code
    - Can be executed/called many times
  - Can have **parameters** to **pass data** into a function
    - **Parameters** have **no explicit types**
    - **Can have default values** for parameters
  - Can return data as a result using the **return** key word
  - **Names** of Python **functions**, by convention, uses **lowercase** letters and **underscores**

Functions in Python

- **Functions can be used just like any other data.**
- **They can be**
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc.

Syntax of a Python function

- A function is defined using a key word def



No declaration of types of arguments or result

Python Functions

- **A function can have parameters and a return value**

```
def sum(num1, num2):
    return num1+num2

num1 = 5
num2 = 6

total = sum(num1, num2)
print(f"Sum of {num1} and {num2} is {total}")

# => Sum of 5 and 6 is 11
```

Python Functions

- **Can have default value for the parameters**
  - When a function is called without passing the argument, it uses default value

```
def say_hello(name = 'John'):
    print("Hello " + name)

say_hello()              # =>  Hello John

say_hello("Paul")        # =>  Hello Paul
```

Python Functions

- **Functions are first-class objects in Python**
  - Functions can be used just like any other data
  - They can be
    - Arguments to function
    - Return values of functions
    - Assigned to variables
    - Parts of tuples, lists, etc

```
def square(x):
    return x**2

def compute(square, x):
    return square(x)

compute(square, 10)     # => returns 100
```

Built-in Python Functions

- **Useful built-in functions**
  - min()
  - max()
  - enumerate()
  - join() method of string
  - isinstance(object, classinfo)
    - checks if the object (first argument) is an instance or subclass of classinfo class (second argument).
  - type(object) -- returns type of the given object

Built-in Python Functions

- **Enumerate() takes a collection (e.g., a tuple) and returns it as an enumerate object**
  - Adds a counter as the key of the enumerate object
- **Syntax: enumerate(*iterable*, *start*), where**
  - *iterable* is an iterable object
  - *start* is the start number of the enumerate object.
    - Default 0

```
fruit = ('apple', 'banana', 'mango')

enumerated_fruit = enumerate(fruit)

print (list(enumerated_fruit))     #   =>   [(0, 'apple'), (1, 'banana'), (2, 'mango')]
```

```
greetings = ['Hello', 'Hi', 'Hey']

x = enumerate(greetings)

print(list(x))       # =>   [(0, 'Hello'), (1, 'Hi'), (2, 'Hey')]
```

Scopes in Python

- **LEGB Scoping Rule**
  - LEGB stands for Local, enclosing, global, and Built-in
  - Python searches a variable in this order

Classes in Python

- **Class – a blueprint or template to define objects**
  - A class definition groups:
    - data (e.g., attributes or member variables) and
    - methods associated with the data
  - *self* keyword is used to assign attributes to class objects inside the class, with *self.* prefix
    - *E.g., self.variable2, self.variable2*
      - *(equivalent to instance variables in Java class)*
  - First parameter of any class method is *self*, referring to this class object
  - Use **. operator** on objects to call object attributes or methods
    - *E.g., object.variable1, object.variable2*

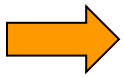
Classes in Python

- **Example: A Shape class with an attribute called 'name' and a method 'draw()**
  - __init__() method is like a constructor in Java
    - used to initialize objects when an object is created

```
import math

class Shape():
    def __init__(self, name):
        self.name = name
        print("Shape created")

    def draw(self):
        print(f'Drawing********** {self.name}')
```

Like instance variables in Java

```
shape = Shape("Shape1")

shape.draw()
```



```
Shape created
Drawing********** Shape1
```

Classes in Python

- **Rectangle class that inherits Shape**
  - Includes additional attributes and methods

```
class Rectangle(Shape):
    #No overloading allowed
    def __init__(self, name, length, width):
        Shape.__init__(self, name)
        self.length = length
        self.width = width
        print(f"{self.name} created")

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2.0*(self.length + self.width)

    #String representation of Rectangle object
    def __repr__(self):
        return f"Rectangle: length = {self.length}, width = {self.width}"
```

Like a Constructor in Java

```
r = Rectangle('Small rectangle', 8, 6)
print(r.draw())
print("Area :" + str(r.area()))
print("Perimeter : " + str(r.perimeter()))
print(r)
```

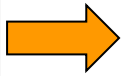
```
Shape created
Small rectangle created
Drawing********** Small rectangle
None
Area :48
Perimeter : 28.0
Rectangle: length = 8, width = 6
```

Classes in Python

- **Circle class that inherits Shape**
  - __repr__() defines string representation of the class object
    - Used to print class objects by print() method
    - Similar to toString() method in Java

```
class Circle(Shape):
    def __init__(self, name, radius):
        Shape.__init__(self, name)
        self.radius = radius
        print(f'{self.name} created')

    def area(self):
        return math.pi*(self.radius **2)

    def perimeter(self):
        return 2.0*math.pi*self.radius

    #String representation of Circle object
    def __repr__(self):
        return f"Circle: radius = {self.radius}"
```

```
c = Circle('Small Circle', 7)
c.draw()
print('Circle area: ' + str(c.area()))
print('Circle perimeter ' + str(c.perimeter()))
print(c)
```



```
Shape created
Small Circle created
Drawing********** Small Circle
Circle area: 153.93804002589985
Circle perimeter 43.982297150257104
Circle: radius = 7
```

Modules and Packages

- **A module is a single file (or files) that are imported under one import and used. e.g.**
  - import my_module
- **A package is a collection of modules in directories that give a package hierarchy**
  - from my_package.timing.danger.internets import function_of_love

Modules and Packages

- **In Python, you can't use an undefined name without defining or importing it first.**
  - For example, you can't call the math module's sin function without importing it.
- **Imports typically go at the very top of your script or Jupyter Notebook.**
  - If you see an unfamiliar name that isn't built in, then it's nearly always the result of an import at the top of the file.

Modules and Packages

- **Three typical ways to import in Python -**
  whatever comes after the keyword import is the name that will be defined in your code.
  - `import math` This imports everything in the `math` module under the name `math`
    - To access any math functions, you would call them using the method notation (e.g., `math.sin(0)`).
  - `from math import sin` This imports only the `sin` function from the `math` module.
    - To use the `sin` function, simply call `sin(0)`
  - `from math import *` This imports every function in the math module as its own name.
    - This is considered bad practice, because it's unclear which names have been added by just looking at the import statement.