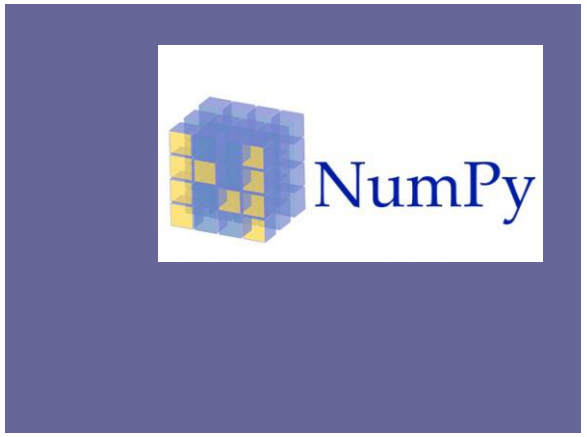


CHULA ENGINEERING
Foundation toward Innovation

COMPUTER

Chula Big Data and IoT
Center of Excellence
(CUBIC)

Data Mining Group
Machine Intelligence and Knowledge Discovery Lab
Chulalongkorn University

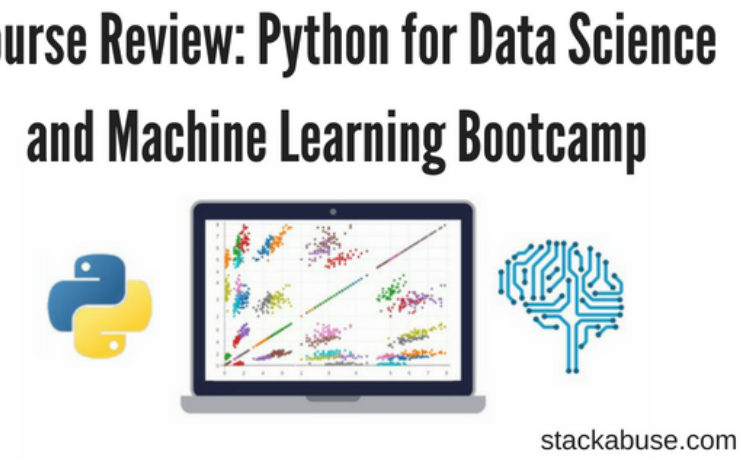
Basic Python and Numpy

Python for Data Analytics

Peerapon Vateekul, Ph.D.
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University
Peerapon.v@chula.ac.th

Reference

- สมชาย ประสิทธิ์จูตระกูล, **2110101 Computer Programming,** ภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย https://www.cp.eng.chula.ac.th/~somchai/
- **Python for Data Science and Machine Learning Bootcamp** https://www.udemy.com/python-for-data-science-and-machine-learning-bootcamp/
- http://cs231n.github.io/python-numpy-tutorial



รศ. ดร. สมชาย ประสิทธิ์จูตระกูล

Outlines

- Section 1: Basic-Python (Overview)
  - Basic data types
  - Operator
  - Input/output
  - Containers
    - Lists
    - Dictionaries
    - Sets
    - Tuples
  - Condition
  - Loop
  - Functions
  - Lambda Function in Python
  - Classes

- Section 2: Numpy
  - Arrays
  - Array indexing
  - Data types
  - Math operation
  - Broadcasting

Reference: http://cs231n.github.io/python-numpy-tutorial/

Section 1

BASIC-PYTHON

Basic-Python (Overview)

- Python is a high-level, dynamically typed multiparadigm programming language.
- Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable.
- As an example, here is an implementation of the classic quicksort algorithm in Python:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
# Prints "[1, 1, 2, 3, 6, 8, 10]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

(Optional) Python versions

- There are currently two different supported versions of Python, 2.7 and 3.6. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.6 and vice versa. **For this class all code will use Python 3.6.3**.

- You can check your Python version at the command line by running
  - python --version.

Reference: http://cs231n.github.io/python-numpy-tutorial/

Basic data types

- Like most languages, Python has a number of basic types including integers, floats, booleans, and strings. These data types behave in ways that are familiar from other programming languages.

- **Numbers**: Integers and floats work as you would expect from other languages:

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Basic data types (cont.)

- Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.
- Python also has built-in types for complex numbers; you can find all of the details in the documentation.
- **Booleans:** Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Basic data types (cont.)

- **Strings:** Python has great support for strings:

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)  # prints "hello world 12"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Basic data types (cont.)

- **String objects** have a bunch of useful methods; for example:

```
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints "  hello"
print(s.center(7))     # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another;
                                # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"
```

You can find a list of all string methods in the documentation. (https://docs.python.org/3.5/library/stdtypes.html#string-methods)

Reference: http://cs231n.github.io/python-numpy-tutorial/

Operators

```
>>> 7+3
10
>>> 7-3
4
>>> 7*3
21
>>> 7/3
2.3333333333333335
>>> 7//3
2
>>> 7%3
1
>>> 2**10
1024
>>> 1.44**0.5
1.2
```

+	บวก
-	ลบ
*	คูณ
/	หาร
//	หารปัดเศษ
%	เศษจากการหาร
**	ยกกำลัง

Reference: 2110101 Computer Programming, Chulalongkorn University.

Operators (cont.)

```
>>> r = 0
>>> r = r + 100
>>> r = r - 30
>>> r = r / 2
>>> r = r // 3
>>> r = r % 7
>>> r = r ** 2
>>> print(r)
.....
```

```
>>> r = 0
>>> r += 100
>>> r -= 30
>>> r /= 2
>>> r //= 3
>>> r %= 7
>>> r **= 2
>>> print(r)
.....
```

แบบไหนเข้าใจง่ายกว่า ?

Input/output

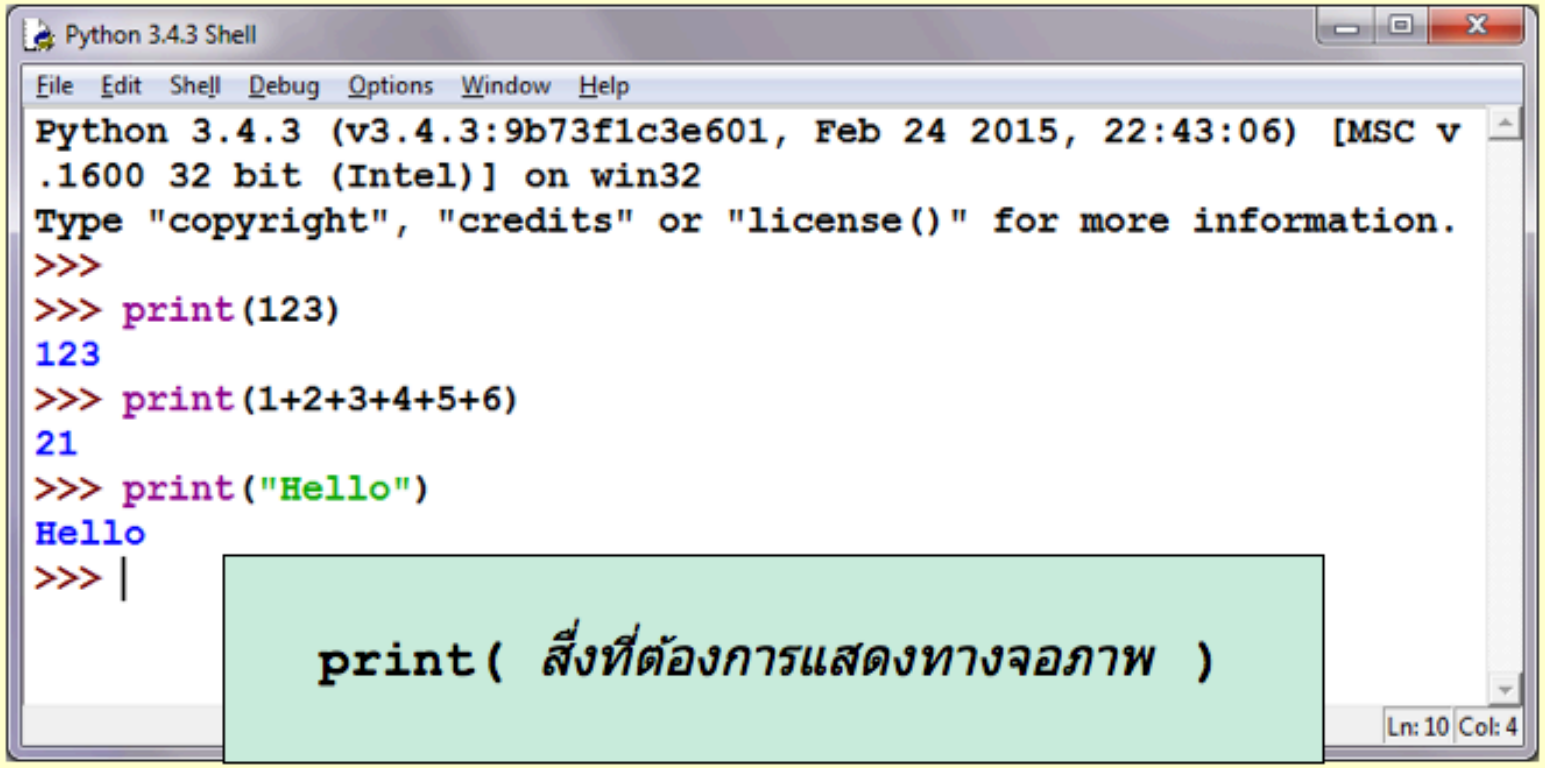
- **input()**
- print()

ใช้ฟังก์ชัน `input( ... )`
ผลที่ได้จาก `input` เป็นสตริง

```
>>> r = input("Enter radius : ")
```

Input/output (cont.)

- input()
- **print()**



print(*สิ่งที่ต้องการแสดงทางจอภาพ*)

ลองดู

```
print(1,2,3)
print("1,2,3",4)
```

Input/output (cont.)

- **input()**
- **print()**

```
>>> r = float( input("Enter radius : ") )
Enter radius : 10
>>> area = 22/7 * r ** 2
>>> print("Area =", area)
Area = 314.2857142857143
>>>
```

แบบนี้ง่ายกว่า

(Optional) Input/output (cont.)

- **input()**
- **print()**

- ถ้ารับข้อมูลจากแป้นพิมพ์ใส่ตัวแปร 1 ตัวในหนึ่งบรรทัด
  - รับสตริง : x = input("") หรือ x = input("").strip()
  - รับจำนวนเต็ม : x = int(input(""))
  - รับจำนวนจริง : x = float(input(""))
- ถ้ารับข้อมูลจากแป้นพิมพ์มากกว่าหนึ่งตัวในหนึ่งบรรทัด (ป้อนข้อมูลแต่ละตัวคั่นด้วยช่องว่าง) `14.5 3.5 4.9`
  - รับสตริง : a, b = [e for e in input("").split()]
  - รับจำนวนเต็ม : x,y,z = [int(e) for e in input("").split()]
  - รับจำนวนจริง : w,h = [float(e) for e in input("").split()]

รายละเอียดจะอธิบายในภายหลัง จำไปก่อน

สำหรับแต่ละตัว e ที่ได้มาจากการแยก input ออกเป็นส่วน ๆ (คั่นด้วยช่องว่าง) แล้วนำ e ไปแปลงเป็น float
ถ้าป้อน 2 จำนวน ต้องมีตัวแปรรับ 2 ตัว
(ถ้าจำนวนที่ป้อนกับจำนวนตัวแปรที่รับไม่ตรงกัน เจ๊ง !!)

Reference: 2110101 Computer Programming, Chulalongkorn University.

(Optional) Python Reserved Words

```
and       as        assert    break     class
continue  def       del       elif      else
except    exec      finally   for       from
global    if        import    in        is
lambda    nonlocal  not       or        pass
raise     return    try       while     with
yield     True      False     None
```

Reference: 2110101 Computer Programming, Chulalongkorn University.

(Optional) Casting and Type Conversions

```
>>> i = 3
>>> f = 3.0
>>> s = "3"
>>> t = "3.0"
>>> i += int(f)      # เปลี่ยน float เป็น int
>>> i += int(s)      # เปลี่ยนสตริงเป็น int
>>> f += float(t)    # เปลี่ยนสตริงเป็น float
>>> s += str(i)      # เปลี่ยน int เป็นสตริง
>>> s = s + str(f)   # เปลี่ยน float เป็นสตริง
>>> print(s)
```

Containers

- Python includes several built-in container types: lists, dictionaries, sets, and tuples.
- **Lists**
  - A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Lists (cont.)**
- **Slicing:** In addition to accessing list elements one at a time, Python provides

```
nums = list(range(5))     # range is a built-in function that creates a list of integer
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])           # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive); prints "
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Lists (cont.)**
- We will see slicing again in the context of numpy arrays.
- **Loops:** You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Lists (cont.)**
- If you want access to the index of each element within the body of a loop, use the built-in **enumerate** function:

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **List comprehensions:** When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

- You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **List comprehensions: (cont.)**
- 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)  # Prints "[0, 4, 16]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Tuples**

- A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)        # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)])  # Prints "1"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Dictionaries**

- A dictionary stores (key, value) pairs, similar to a **Map** in Java or an object in Javascript. You can use it like this:

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Dictionaries (cont.)**
- **Loops:** It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- If you want access to keys and their corresponding values, use the **items** method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Dictionaries (cont.)**
- **Dictionary comprehensions:** These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
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)  # Prints "{0: 0, 2: 4, 4: 16}"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Sets**

- A set is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')        # Adding an element that is already in the set does nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.)

- **Sets (cont.)**

- **Loops:** Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

- **Set comprehensions:** Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Containers (cont.) (Summary in Thai)

	list	tuple	dict	set
การใช้	- ลำดับของข้อมูลมีความหมาย อาจมีการเปลี่ยนแปลง - ข้อมูลในรายการมักมีความหมายเดียวกัน	- ลำดับของข้อมูลมีความหมาย - สร้างแล้วไม่เปลี่ยนแปลง - ข้อมูลใน tuple มักมีความหมายต่างกัน	- เก็บข้อมูลเป็นคู่ๆ key-value โดยใช้ key เข้าถึงข้อมูลเพื่อให้ได้ value มาใช้งาน	- เก็บข้อมูลไม่ซ้ำ ลำดับของข้อมูลไม่มีความหมาย เพื่อตรวจสอบว่า มีข้อมูลหรือไม่ รองรับ set operations
การเข้าใช้ข้อมูล	ใช้จำนวนเต็มระบุตำแหน่ง `d[i]`	ใช้จำนวนเต็มระบุ `d[i]`	ใช้ key เป็นตัวระบุตำแหน่งข้อมูล `d[key]`	ต้อง `for...in...` เพื่อแจงข้อมูล
การค้นด้วย `in`	ค้นจากซ้ายไปขวาช้า	ค้นจากซ้ายไปขวาช้า	มีวิธีค้นที่เร็วมาก	มีวิธีค้นที่เร็วมาก
การสร้าง	`x =[1,2,3,4]`	`t = (1,2,3,4)`	`d = { "k1":1, "k2":2 }`	`s = {1,2,3,4}`

การเพิ่มข้อมูล	`x.append(3)`
	`x.insert(1,99)`
		สร้างแล้วเปลี่ยนแปลงไม่ได้ ต้องสร้างใหม่
		`t = t + (4, )`
			`d["k1"] = 1`
			`d["k2"] = 2`
			หรือใช้ `update`
				`s.add(3)`

Reference: **2110101 Computer Programming, Chulalongkorn University.**

Condition

- **What are if…else statement in Python?**
  - Decision making is required when we want to execute a code only if a certain condition is satisfied.
  - The if…elif…else statement is used in Python for decision making.
- **Python if Statement Syntax**

```
if test expression:
    statement(s)
```

  - Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True.
  - If the text expression is False, the statement(s) is not executed.
  - In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end.
  - Python interprets non-zero values as True. None and 0 are interpreted as False.

Reference: https://www.programiz.com/python-programming/if-elif-else

Condition (cont.)

- **Python if Statement Flowchart**

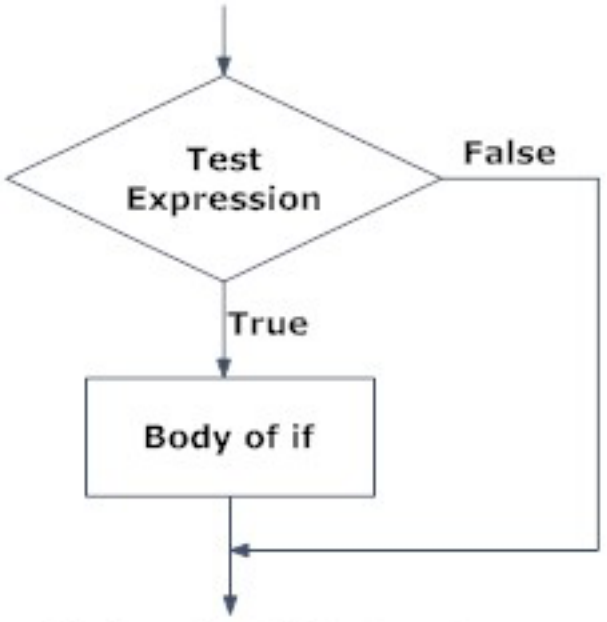


Fig: Operation of if statement

- **Example: Python if Statement**

script.py | IPython Shell

```
1   # If the number is positive, we print an appropriate message
2
3   num = 3
4   if num > 0:
5       print(num, "is a positive number.")
6   print("This is always printed.")
7
8   num = -1
9   if num > 0:
10      print(num, "is a positive number.")
11  print("This is also always printed.")
```

OUTPUT

```
3 is a positive number.
This is always printed.
This is also always printed.
```

Reference: https://www.programiz.com/python-programming/if-elif-else

Condition

- **Python if…else Statement**
- **Syntax of if…else**

```
if test expression:
     Body of if
else:
     Body of else
```

- The if..else statement evaluates test expression and will execute body of if only when test condition is True.
- If the condition is False, body of else is executed. Indentation is used to separate the blocks.

Reference: https://www.programiz.com/python-programming/if-elif-else

Condition (cont.)

- **Python if…else Statement**
- **Syntax of if…else**

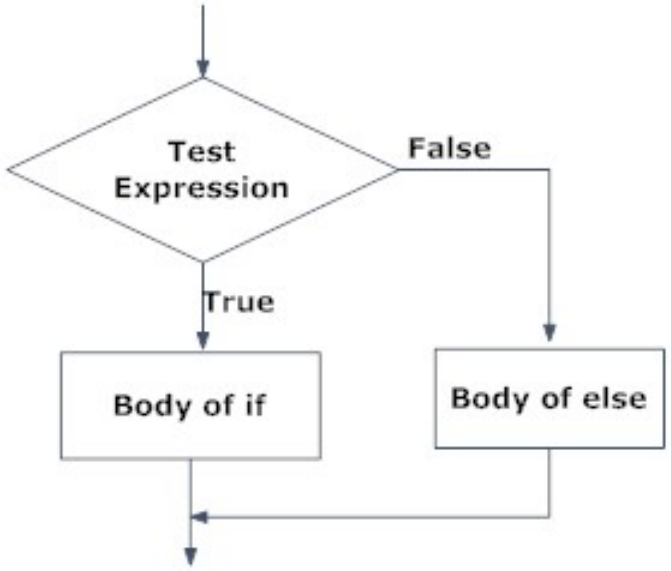


Fig: Operation of if...else statement

- **Example: Python if Statement**

script.py | IPython Shell

```
1   # In this program,
2   # we check if the number is positive or
3   # negative or zero and
4   # display an appropriate message
5
6   num = 3.4
7
8   # Try these two variations as well:
9   # num = 0
10  # num = -4.5
11
12  if num > 0:
13      print("Positive number")
14  elif num == 0:
15      print("Zero")
16  else:
17      print("Negative number")
```

OUTPUT

Positive Number

Reference: https://www.programiz.com/python-programming/if-elif-else

Condition (cont.)

- **Python Nested if statements**
- We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.
- Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

OUTPUT

Output 1

```
Enter a number: 5
Positive number
```

Output 2

```
Enter a number: -1
Negative number
```

Output 3

```
Enter a number: 0
Zero
```

Reference: https://www.programiz.com/python-programming/if-elif-else

Loop

- **What is for loop in Python?**
- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.
- **Syntax of for Loop**

```
for val in sequence:
        Body of for
```

- Here, val is the variable that takes the value of the item inside the sequence on each iteration.
- Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Reference: https://www.programiz.com/python-programming/for-loop

Loop (cont.)

- **Flowchart of for Loop**

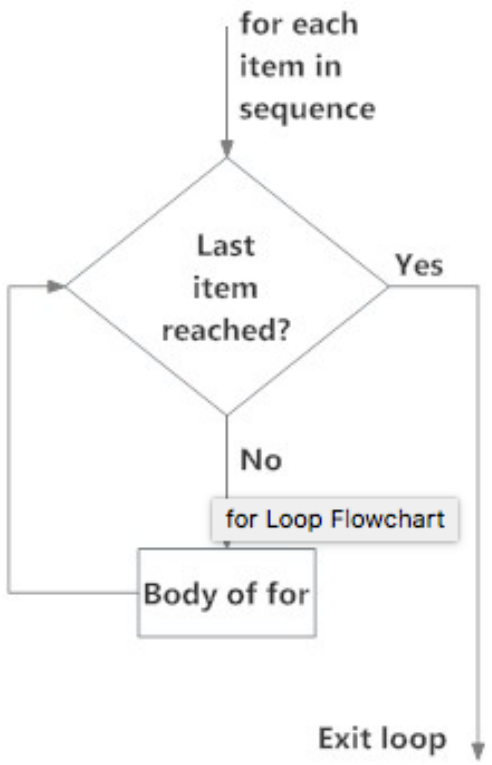


Fig: operation of for loop

- **Example: Python for Loop**

script.py | IPython Shell

```
1   # Program to find the sum of all numbers stored in a list
2
3   # List of numbers
4   numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
5
6   # variable to store the sum
7   sum = 0
8
9   # iterate over the list
10  for val in numbers:
11      sum = sum+val
12
13  # Output: The sum is 48
14  print("The sum is", sum)
```

The sum is 48

Reference: https://www.programiz.com/python-programming/for-loop

Reference: https://www.programiz.com/python-programming/for-loop

Loop (cont.)

- **The range() function**
- We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.
- This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.
- To force this function to output all the items, we can use the function list().
- The following example will clarify this.

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))
```

```
# Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))
```

Loop (cont.)

- We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing. Here is an example.

script.py | IPython Shell

```
1  # Program to iterate through a list using indexing
2
3  genre = ['pop', 'rock', 'jazz']
4
5  # iterate over the list using index
6  for i in range(len(genre)):
7      print("I like", genre[i])
```

OUTPUT

```
I like pop
I like rock
I like jazz
```

Reference: https://www.programiz.com/python-programming/for-loop

Loop (cont.)

- **What is while loop in Python?**
- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- We generally use this loop when we don't know beforehand, the number of times to iterate.
- **Syntax of while Loop in Python**

```
while test_expression:
    Body of while
```

Reference: https://www.programiz.com/python-programming/for-loop

Loop (cont.)

- **Flowchart of for Loop**

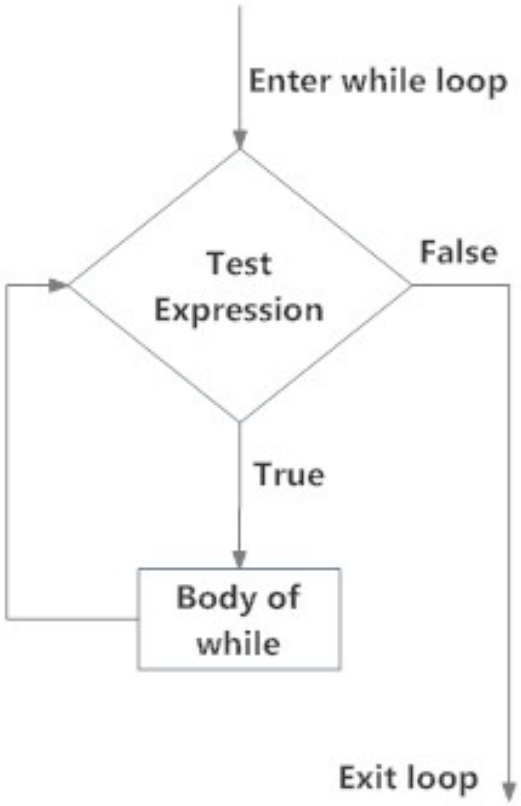


Fig: operation of while loop

- **Example: Python for Loop**

script.py | IPython Shell

```
1   # Program to add natural
2   # numbers upto
3   # sum = 1+2+3+...+n
4
5   # To take input from the user,
6   # n = int(input("Enter n: "))
7
8   n = 10
9
10  # initialize sum and counter
11  sum = 0
12  i = 1
13
14  while i <= n:
15      sum = sum + i
16      i = i+1    # update counter
17
18  # print the sum
19  print("The sum is", sum)
```

```
Enter n: 10
The sum is 55
```

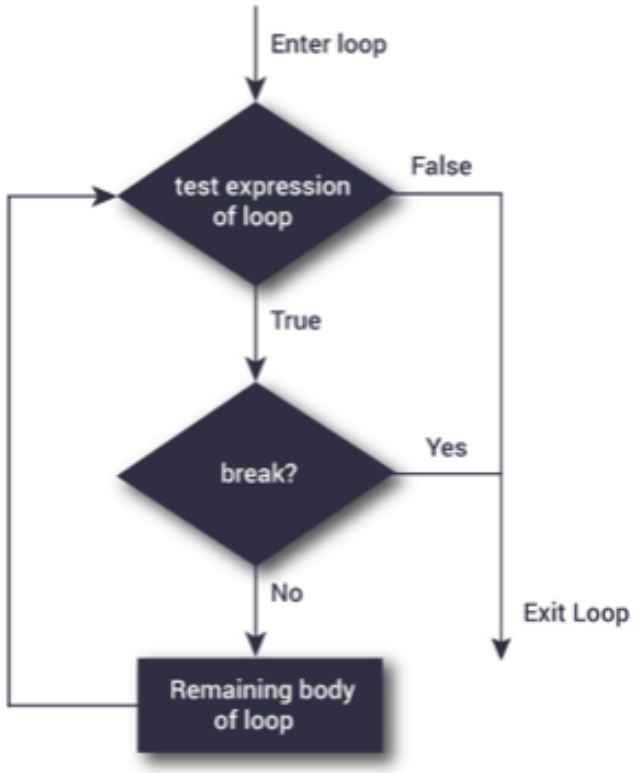
Reference: https://www.programiz.com/python-programming/for-loop

Loop (cont.) – Break

- **What is the use of break and continue in Python?**
  - In Python, break and continue statements can alter the flow of a normal loop.
  - Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.
  - The break and continue statements are used in these cases.
- **Python break statement**
  - The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.
  - If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Loop (cont.) – Break (cont.)

- **Flowchart of for Loop**



- **Example: Python for Loop**

script.py | IPython Shell

```
1  # Use of break statement inside loop
2
3  for val in "string":
4      if val == "i":
5          break
6      print(val)
7
8  print("The end")
```

```
s
t
r
The end
```

Reference: https://www.programiz.com/python-programming/for-loop

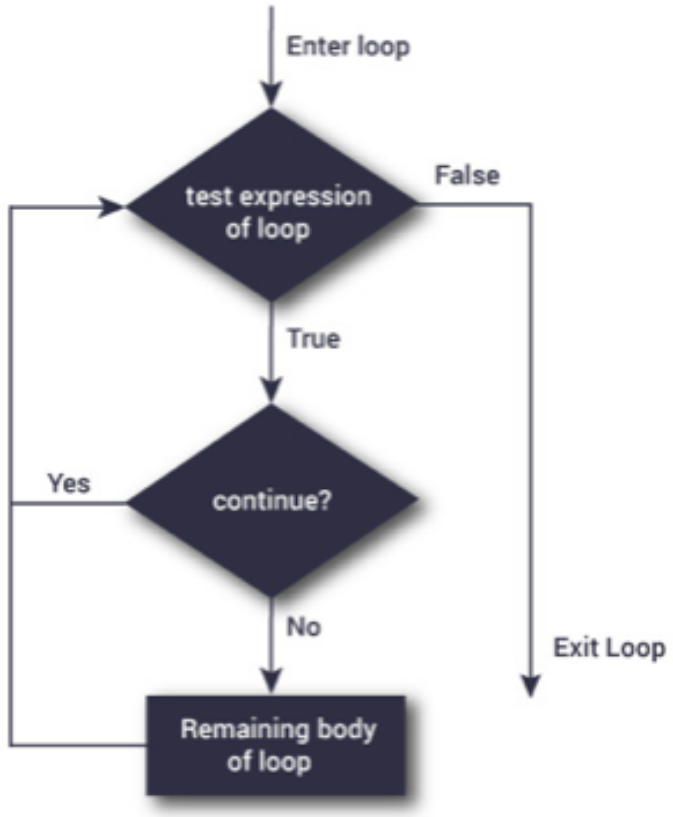
Loop (cont.) – Continue

- **Python continue statement**
  - The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.
- **Syntax of Continue**

```
continue
```

Loop (cont.) – Continue(cont.)

- **Flowchart of for Loop**



- **Example: Python for Loop**

script.py | IPython Shell

```
1  # Program to show the use of continue statement inside loops
2
3  for val in "string":
4      if val == "i":
5          continue
6      print(val)
7
8  print("The end")
```

```
s
t
r
n
g
The end
```

Reference: https://www.programiz.com/python-programming/for-loop

Functions

- Python functions are defined using the **def** keyword. For example:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Functions (cont.)

- We will often define functions to take optional keyword arguments, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Functions (cont.)

- We will often define functions to take optional keyword arguments, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Lambda Function in Python

- **What are lambda functions in Python?**
- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions.

Reference: https://www.programiz.com/python-programming/anonymous-function

Lambda Function in Python (cont.)

- **How to use lambda Functions in Python?**
- A lambda function in python has the following syntax.
- **Syntax of Lambda Function in python**

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Reference: https://www.programiz.com/python-programming/anonymous-function

Lambda Function in Python (cont.)

- In the above program, lambda x: x * 2 is the lambda function. Here x is the argument and x * 2 is the expression that gets evaluated and returned.
- This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

- is nearly the same as

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

Reference: https://www.programiz.com/python-programming/anonymous-function

Lambda Function in Python (cont.)

- **Use of Lambda Function in python**
- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like **filter(), map()** etc.

Reference: https://www.programiz.com/python-programming/anonymous-function

Lambda Function in Python (cont.)

- **Example use with filter()**
- The **filter()** function in Python takes in a function and a list as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.
- Here is an example use of filter() function to filter out only even numbers from a list.

script.py IPython Shell

```
1  # Program to filter out only the even items from a list
2
3  my_list = [1, 5, 4, 6, 8, 11, 3, 12]
4
5  new_list = list(filter(lambda x: (x%2 == 0) , my_list))
6
7  # Output: [4, 6, 8, 12]
8  print(new_list)
```

Reference: https://www.programiz.com/python-programming/anonymous-function

Lambda Function in Python (cont.)

- **Example use with map()**
- The **map()** function in Python takes in a function and a list.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.
- Here is an example use of map() function to double all the items in a list.

script.py IPython Shell

```
1  # Program to double each item in a list using map()
2
3  my_list = [1, 5, 4, 6, 8, 11, 3, 12]
4
5  new_list = list(map(lambda x: x * 2 , my_list))
6
7  # Output: [2, 10, 8, 12, 16, 22, 6, 24]
8  print(new_list)
```

Reference: https://www.programiz.com/python-programming/anonymous-function

Classes

- The syntax for defining classes in Python is straightforward:

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Section2

NUMPY

Numpy

- Numpy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.
- If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

Reference: http://cs231n.github.io/python-numpy-tutorial/

Arrays

- A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

- We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Arrays (cont.)

- Numpy also provides many functions to create arrays:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Reference: http://cs231n.github.io/python-numpy-tutorial/

Array indexing

- Numpy offers several ways to index into arrays.

- **Slicing:** Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Array indexing (cont.)

- You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Array indexing (cont.)

- **Integer array indexing:** When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

Array indexing (cont.)

- One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"
```

```
# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Reference: http://cs231n.github.io/python-numpy-tutorial/

Array indexing (cont.)

- **Boolean array indexing:** Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Data types – Numpy

- Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Math operation – Numpy

- Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Math operation – Numpy (cont.)

- Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Math operation – Numpy (cont.)

- Numpy provides many useful functions for performing computations on arrays; one of the most useful is sum:

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Math operation – Numpy (cont.)

- Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the **T** attribute of an array object:

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

- For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Broadcasting (cont.)

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

- For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this: **(cont.)**

```
# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Broadcasting (cont.)

- This works; however when the matrix **x** is very large, computing an explicit loop in Python could be slow. Note that adding the vector **v** to each row of the matrix **x** is equivalent to forming a matrix **vv** by stacking multiple copies of **v** vertically, then performing elementwise summation of **x** and **vv**. We could implement this approach like this:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Broadcasting (cont.)

- Numpy broadcasting allows us to perform this computation without actually creating multiple copies of **v**. Consider this version, using broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

- The line **y = x + v** works even though **x** has shape **(4, 3)** and v has shape **(3,)** due to broadcasting; this line works as if **v** actually had shape **(4, 3)**, where each row was a copy of **v**, and the sum was performed elementwise.

Reference: http://cs231n.github.io/python-numpy-tutorial/

Broadcasting (cont.)

- Here are some applications of broadcasting:

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Broadcasting (cont.)

- Here are some applications of broadcasting **(cont.):**

```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

Reference: http://cs231n.github.io/python-numpy-tutorial/

Numpy Documentation

- This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the **numpy reference** to find out much more about numpy.

https://docs.scipy.org/doc/numpy/reference/

Reference: http://cs231n.github.io/python-numpy-tutorial/

Lab: Basic Python and Numpy

Python Crash Course

1) Basic Data Types
- Data types
- Variable Assignment
- Strings
- Printing
- Input

Hand-on

2) Collections
- Lists
- List Slicing

Example: various list operations
- Dictionaries
- Tuples
- Sets
- Booleans

3) Operators
- Comparison Operators
- Logic Operators

4) Condition
- if,elif, else Statements

5) Loops
- for Loops
- while Loops
- range()
- list comprehension
- EXAMPLE: perfect squares

6) Functions
- functions
- EXAMPLE: returning a tuple
- Fibonacci numbers
- types

7) Lambda Exp
- ambda expressions
- map and filter

8) Class
- Class
- methods

9) Import Lib
- import library

- **Numpy**
- 1) Overview
  - Types: Vectors & Matrices
  - Indexing
  - Create data
- 2) Operations
  - +, -, *, /, etc.
  - Method

+

Any Questions?