CS 412 Introduction to Machine Learning

Python for Machine Learning

Instructor: Wei Tang

Department of Computer Science
University of Illinois at Chicago
Chicago IL 60607

https://tangw.people.uic.edu
tangw@uic.edu

Slides credit: Chariza Pladin, Justin Johnson

Python

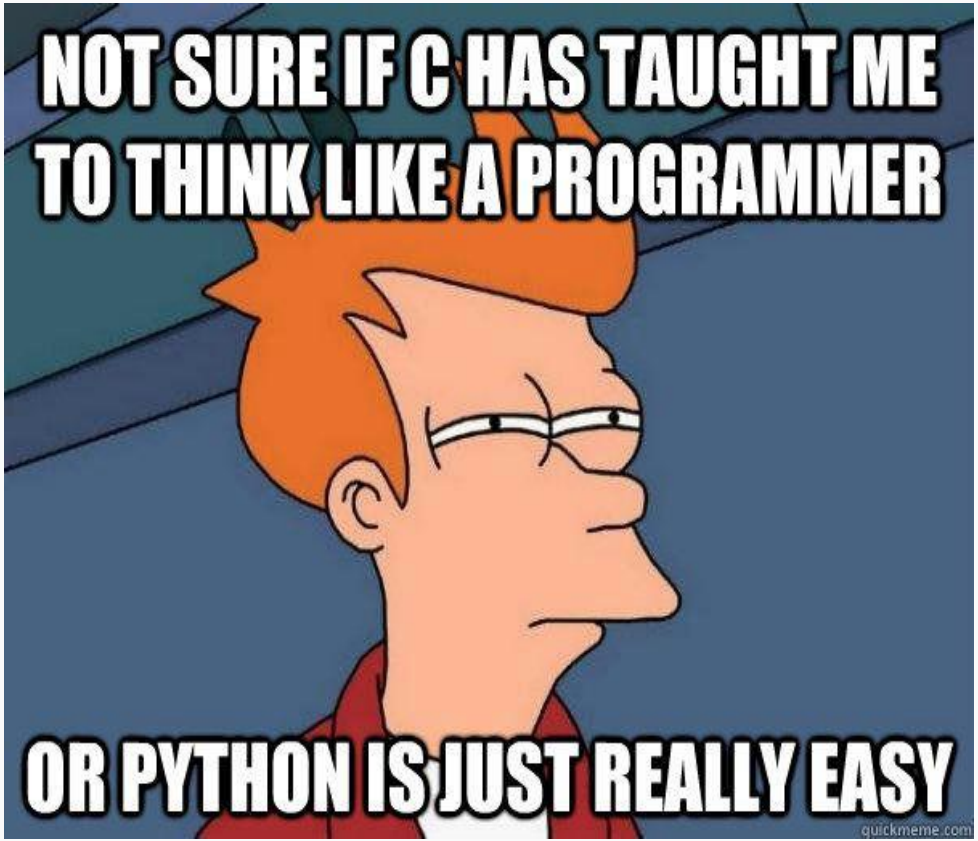
High-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991.

Python is great for backend web development, data analysis, artificial intelligence, and scientific computing.

Many developers have also used Python to build productivity tools, games, and desktop apps.

Why learn Python?

Great for Beginners



Simple
Elegant
Syntax

Python takes coding like natural human-language.

```
a = 10
b = 12

sum = a + b
print (sum)
```

Not overly Strict

You don't need to define the type of a variable in Python.

```
a = 10
b = "Ten"
c = 0.10
d = -12
e = False
```

Expressive of Language

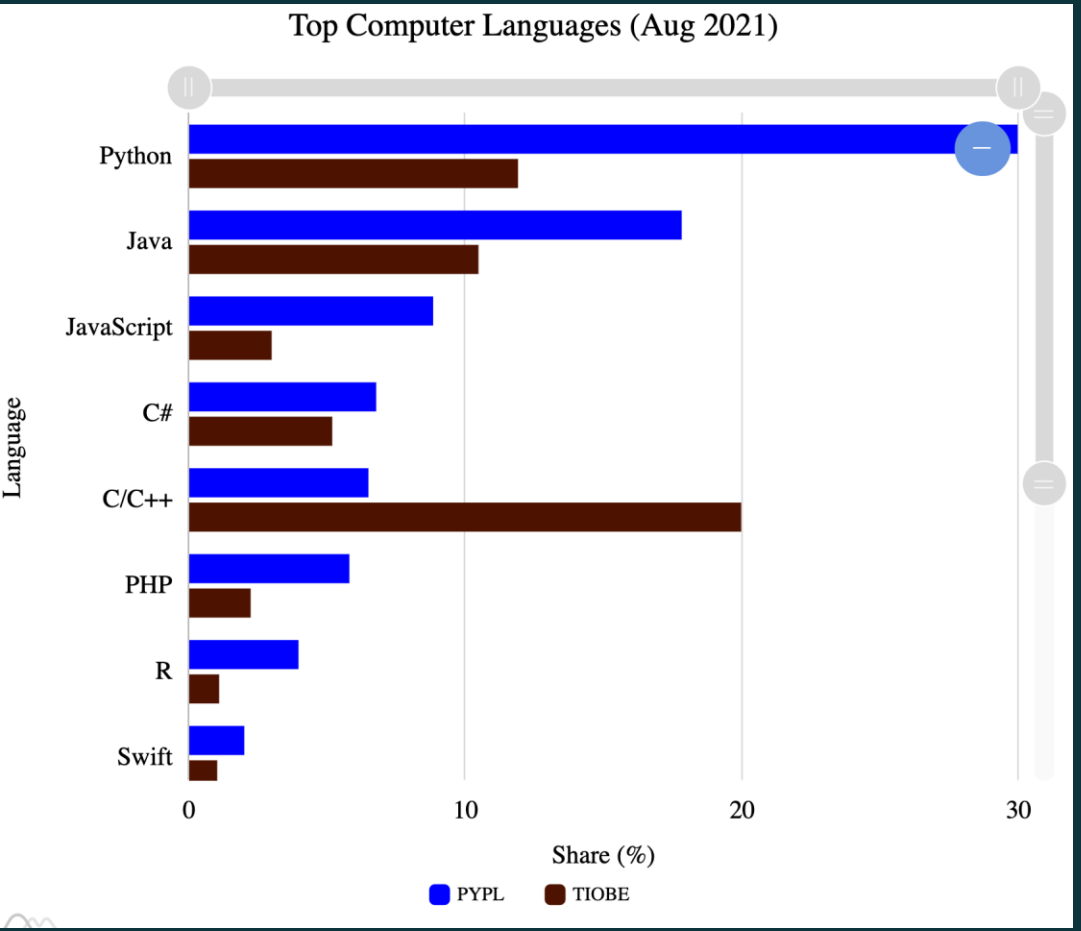
Python allows you to write programs having greater functionality with fewer lines of code.

```
def getCount(inputStr):
  '''
  return vowel count from the given string.
  '''
  return sum(1 for letter in inputStr if letter in
    'aeiouAEIOU')
```

Very Flexible

No hard rules on how to build features, and you'll have more flexibility solving problems using different methods.

Furthermore, Python is also more forgiving of errors, **so you'll still be able to run your program until you hit the problematic part.**

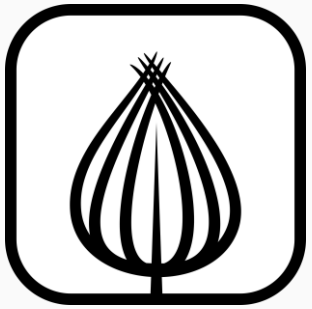


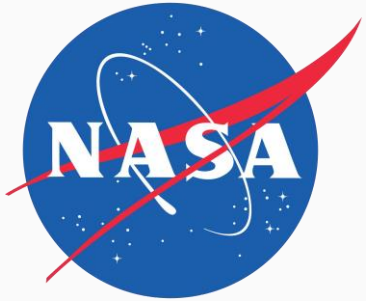
Career Opportunities and the Future

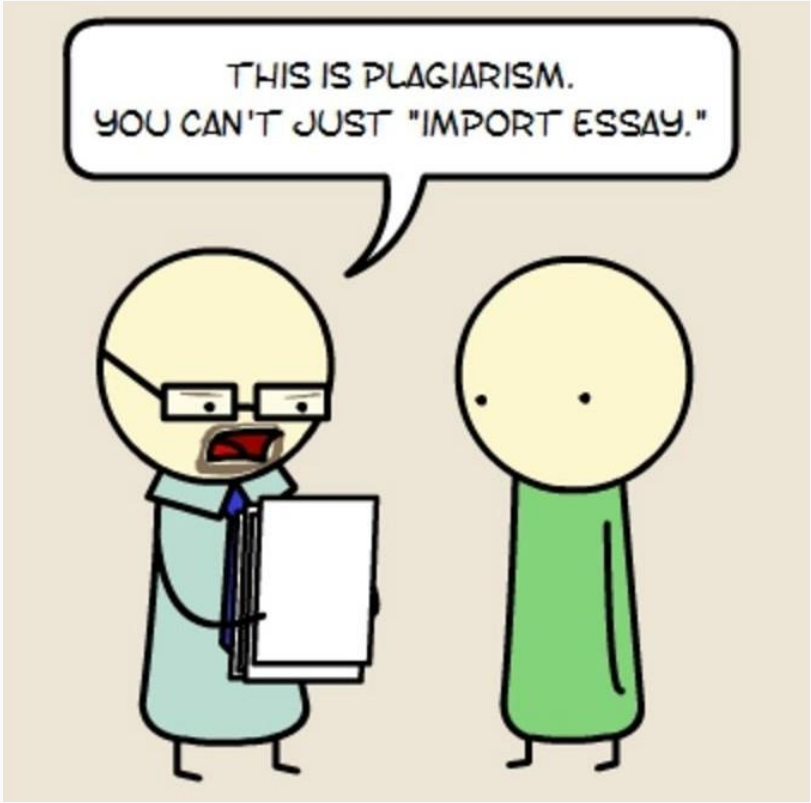
https://statisticstimes.com/tech/top-computer-languages.php

Companies that uses Python

Apps







The import system

Modules importing

```
from math import pi

def getCirc(rad):
  return 2*(pi*(rad ** 2))

''' print the circumference '''
print (getCirc(20))
```

Modules

Modules can define functions, classes, and variables that you can reference in other Python .py files or via the Python command line interpreter.

Modules are accessed by using the **import** statement.

Python Frameworks

DATA SCIENCE

MACHINE LEARNING

GAME DEVELOPMENT

PANDA3D

OPEN SOURCE GAME ENGINE

WEB DEVELOPMENT

A Questionnaire for CS 412 (2021 Fall)

Are you familiar with Python?

42 responses

Are you familiar with Numpy?

42 responses

Basic Syntax

Variable

A **variable** is a location in memory used to store some data (value).

We **don't need** to declare a variable before using it.

We **don't** even have to declare the type of the variable.

This is handled internally according to the type of value we assign to the variable.

Variable Assignment

We use the **assignment operator (=)** to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a_name = "John Doe"
a_age = 12
a_gender = "Male"
a_ave = 84.92
```

Multiple Assignment

In Python, multiple assignments can be made in a single statement as follows:

```
a, b, c = 5, 3.2, "Hello"
```

If we want to assign the same value to multiple variables at once, we can do this as

```
x = y = z = 6
```

Basic Operators

Basic Operators

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Basic Operators

- + Addition
- - Subtraction
- * Multiplication
- / Division
- ** Exponent

Python Comparison Operators

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

Python Assignment Operators

- = Equal
- += Add AND
- -= Subtract AND
- *= Multiply AND
- /= Divide AND
- **= Exponent AND

```
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"
```

Python Logical Operators

- and: logical AND
- or: logical OR
- not: logical NOT

```
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"
```

Python Data Types

Data Types

Numeric Types

- **int:** Integers;
- **float:** Floating-Point numbers, equivalent to C doubles
- **long:** Long integers of non-limited length;
- **complex:** Complex Numbers

Sequences Types

- **str:** String;
- **list**
- **Tuple**
- **Set**
- **dict**
- **bytes:** a sequence of integers in the range of 0-255; only available in Python 3.x
- **byte array:** like bytes, but mutable

List

List is the most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets.

Important thing about a list is that items in a list need not be of the same type.

List

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

Basic List Operations

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Python Identity Operators

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

```
>>> x, y = 1, 1
>>> print(id(x), id(y))
4548217584 4548217584
>>> print(x is y, x is not y)
True False
>>> x, y = [1, 1], [1,1]
>>> print(id(x), id(y))
4550816768 4550816896
>>> print(x is y, x is not y)
False True
>>> y = x
>>> print(id(x), id(y))
4550816768 4550816768
>>> print(x is y, x is not y)
True False
```

Python Membership Operators

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

```
>>> 1 in [1, 2]
True
>>> 3 in [1, 2]
False
>>> 1 not in [1, 2]
False
```

Built-in List Functions

len(list)

Gives the total length of the list.

max(list)

Returns item from the list with max value.

min(list)

Returns item from the list with min value.

list(seq)

Converts a tuple into list.

List Methods

list.append(obj)

Appends object obj to list

list.count(obj)

Returns count of how many times obj occurs in list

list.extend(seq)

Appends the contents of seq to list

list.index(obj)

Returns the lowest index in list that obj appears

list.insert(index, obj)

Inserts object obj into list at offset index

list.pop(obj=list[-1])

Removes and returns last object or obj from list

List Methods (cont.)

list.remove(obj)

Removes object obj from list

list.reverse()

Reverses objects of list in place

list.sort([func])

Sorts objects of list, use compare func if given

```
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']"
```

Tuple

Tuples are immutable which means you cannot update or change the values of tuple elements.

Tuple Examples

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

Basic Tuple Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in Tuple Functions

len(tuple)

Gives the total length of the tuple.

max(tuple)

Returns item from the tuple with max value.

min(tuple)

Returns item from the tuple with min value.

tuple(seq)

Converts a list into tuple.

Dictionary

Each key is separated from its value by a **colon (:)**, the items are separated by commas, and the whole thing is enclosed in **curly braces**.

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
```

Keys are unique within a dictionary while values may not be.

The values of a dictionary can be of **any type**, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}


print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

Updating Values in Dictionary

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Deleting Values in Dictionary

```
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

```
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"
```

Dictionary Function and Methods

dict.clear()

Removes all elements of dictionary *dict*

dict.copy()

Returns a shallow copy of dictionary *dict*

dict.fromkeys()

Create a new dictionary with keys from seq and values *set* to *value*.

dict.get(key, default=None)

For *key* key, returns value or default if key not in dictionary

dict.has_key(key)

Returns *true* if key in dictionary *dict*, *false* otherwise

dict.items()

Returns a list of *dict*'s (key, value) tuple pairs

Dictionary Function and Methods (cont.)

dict.keys()

Returns list of dictionary dict's keys

dict.setdefault(key, default=None)

Similar to get(), but will set dict[key]=default if *key* is not already in dict

dict.update(dict2)

Adds dictionary *dict2*'s key-values pairs to *dict*

dict.values()

Returns list of dictionary *dict*'s values

Built-in Dictionary Functions

len(dict)

Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

str(dict)

Produces a printable string representation of a dictionary

type(variable)

Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Strings

We can create strings simply by enclosing characters in quotes.

Python treats single quotes the same as double quotes.

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing String Values

Python does not support a character type; these are treated as strings of **length one**, thus also considered **a substring.**

```
var1 = 'Hello World!'
var2 = "Python Programming"

print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string.

```
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

```
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"
```

```
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"
```

Built-in String Methods

capitalize()
Capitalizes first letter of string

center(width, fillchar)

Returns a space-padded string with the original string centered to a total of width columns.

count(str, beg= 0,end=len(string))

Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.

decode(encoding='UTF-8',errors='strict')

Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.

Built-in String Methods (cont.)

encode(encoding='UTF-8',errors='strict')

Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.

endswith(suffix, beg=0, end=len(string))

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.

expandtabs(tabsize=8)

Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.

find(str, beg=0 end=len(string))

Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.

Built-in String Methods (cont.)

index(str, beg=0, end=len(string))

Same as find(), but raises an exception if str not found.

isalnum()

Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

isalpha()

Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

isdigit()

Returns true if string contains only digits and false otherwise.

Built-in String Methods (cont.)

islower()

Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

isnumeric()

Returns true if a unicode string contains only numeric characters and false otherwise.

isspace()

Returns true if string contains only whitespace characters and false otherwise.

istitle()

Returns true if string is properly "titlecased" and false otherwise.

Built-in String Methods (cont.)

isupper()

Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.

join(seq)

Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

len(string)

Returns the length of the string

ljust(width[, fillchar])

Returns a space-padded string with the original string left-justified to a total of width columns.

lower()

Converts all uppercase letters in string to lowercase.

Python Decision Making

Decision Making

Statement	Description
if statements	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statements	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Decision Making

```
foo = 10

if foo == 10:
   print ("foo is equals to ",foo)
```

```
foo = 10

if foo == 10:
  print("foo is equals to ",foo)
else:
  print("foo is not equals to",foo)
```

```
foo = 9

if foo == 10:
  print("foo is equals to ",foo)
elif foo > 10:
  print("foo is greater than ",foo)
else:
  print("foo is not equals to",foo)
```

Python Loops

Loops

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for or do..while loop.

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.
```

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
```

Loops Control

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Range()

-returns an immutable sequence object of integers between the given start integer to the stop integer.

Syntax:

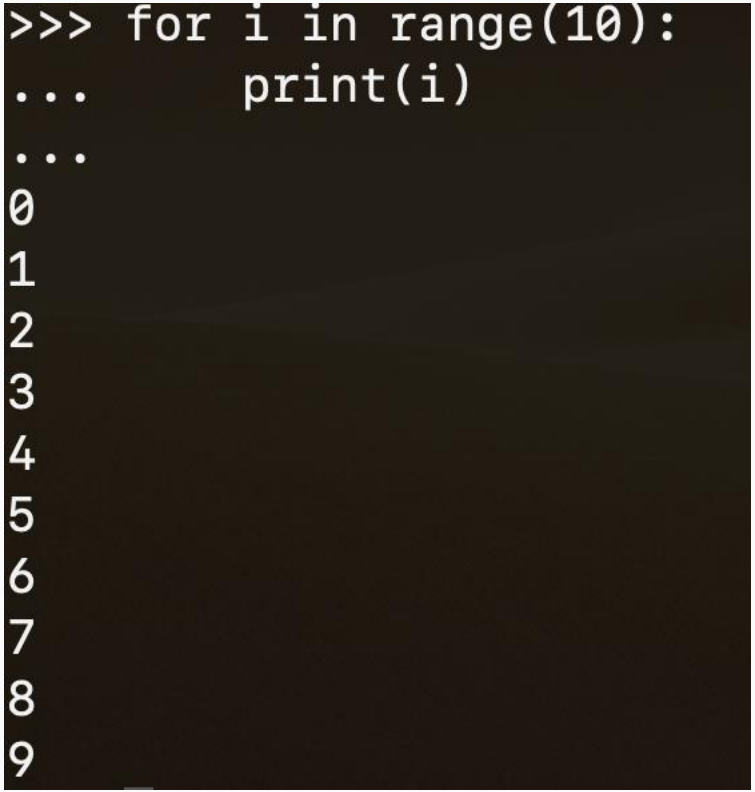
range(stop)

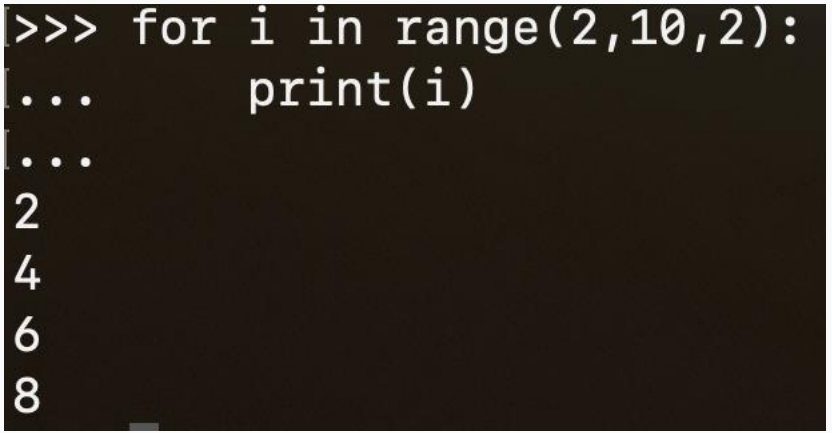
range(start, stop[, step])

Range() Parameters

start - integer starting from which the sequence of integers is to be returned
stop - integer before which the sequence of integers is to be returned.
The range of integers end at stop - 1.

step (Optional) - integer value which determines the increment between each integer in the sequence


>>> for i in range(10):
...   print(i)
...
0
1
2
3
4
5
6
7
8
9


>>> for i in range(2,10,2):
...   print(i)
...
2
4
6
8

Python Functions

Functions

A function is a block of organized, reusable code that is used to perform a single, related action.

Functions provide better modularity for your application and a **high degree of code reusing.**

Function blocks begin with the keyword **def** followed by the function name and parentheses (( ))

Any input parameters or arguments should be placed within these parentheses.

the documentation string of the function or docstring.

```
def functionname( parameters ):
    '''function_docstring'''
    function_suite
    return [expression]
```

statement return [expression] exits a function, optionally passing back an expression to the caller.

Docstrings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.

Such a docstring becomes the __doc__special attribute of that object.

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
```

Comments

Comments are little snippets of text embedded inside your code that are ignored by the Python interpreter.

A comment is denoted by the hash character (#) and extends to the end of the line.:

```
# Display the knights that come after Scene 24
print("The Knights Who Say Ni!")
```

Functions Examples

```
def multiply(a,b):
  '''this function will multiply 2 numbers (a and b) '''
  return a*b
```

```
def getData(uname,pw):
  '''print user's name and censored password'''
  print ('Hello there!',uname,'\nYour password is',len(pw)*('*'))
```

```
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"
```

```
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!"
```

More on Python

https://docs.python.org/3/tutorial/

But the content till now is sufficient for this course.

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.

Arrays

A numpy array is a grid of values, all of the same type

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"
```

Array creation functions

```
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]]"
```

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:

```
# 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"
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array

```
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)"
```

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]"
```

Datatypes

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.

```
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"
```

Array math

Basic mathematical functions operate **elementwise** on arrays

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))
```

Matrix multiplication: dot

```
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))
```

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]"
```

Transpose

```
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]"
```

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.

```
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]]"
```

More on Numpy

https://numpy.org/doc/stable/reference/

But the content till now is sufficient for this course.