PACE UNIVERSITY

SEIDENBERG SCHOOL OF CSIS

python™

NumPy

SciPy.org

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

matplotlib

The Python Official Site
https://www.python.org

The Python Standard Library Documentation
https://docs.python.org/3/library/

Seidenberg School Of Computer Sciences | Pace University

Introduction to Python Programming

NumPy, SciPy, Pandas, Matplotlib, Seaborn, Bokeh
Anaconda Open Data Science Platform

Prof. Tassos H. Sarbanes

CS675 --- Spring 2022

Hot Technology Trends to Track & Learn

On-the-rise technology trends to track and learn

AI, **Python**, **Java**, **blockchain**, and **cloud technologies** are active topics on O'Reilly's online learning platform.

Python remains the No. 1 search term on online learning platform, and its share of search activity is increasing. Python's growth is likely fueled by the language's adoption for data management, data engineering, and data analytics tasks, including machine learning (ML) and artificial intelligence (AI). Python is worth knowing and is a good choice for development, particularly when working with data pipelines.

ML and AI have become topics of keen interest to learners on AI platforms. Terms like **machine learning** (the No. 5 search term and up 42% in year-over-year search activity), **deep learning** (No. 18, up 60%), and **TensorFlow** (No. 19, up 146%), all show considerable growth in activity. While still maturing, we see more learners moving past the exploration of ML and AI, and into the practical application of these tools.

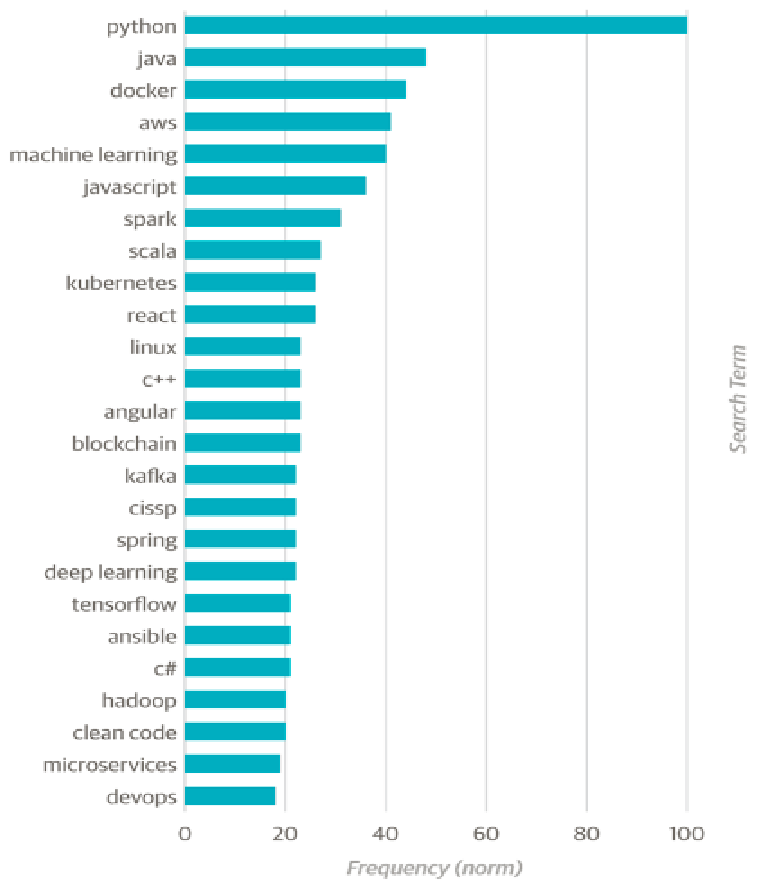
https://www.oreilly.com/ideas/7-on-the-rise-technology-trends-to-track-and-learn

See next slide for details…..

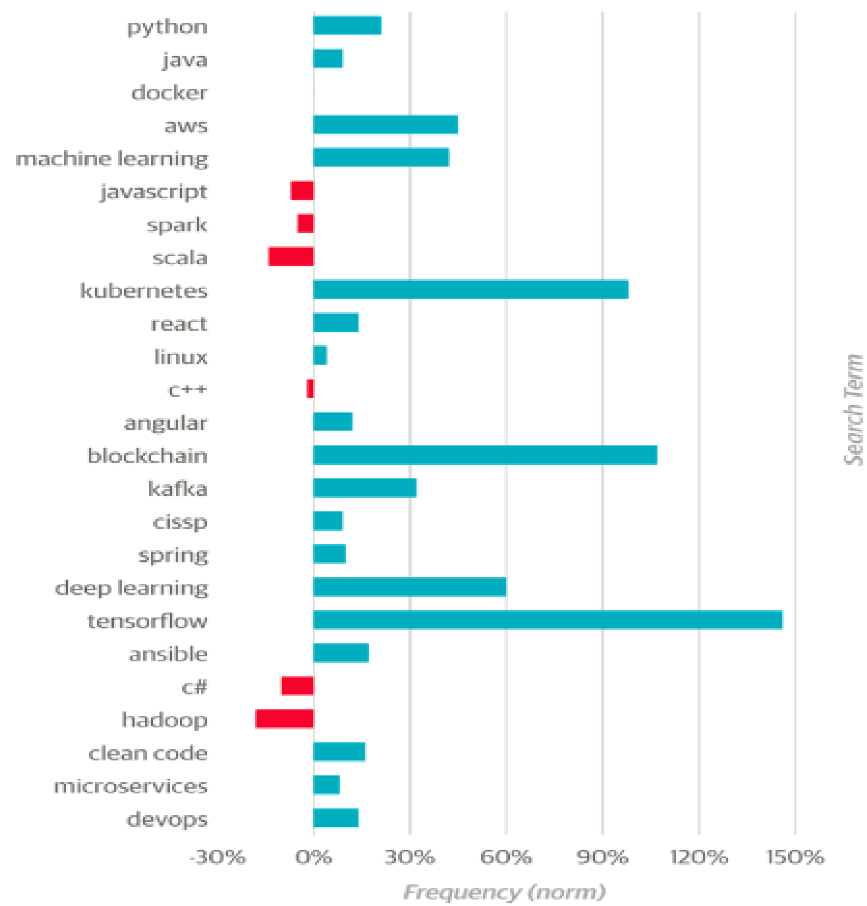
Top Online Learning Platforms (O'Reilly media)

The continued dominance of Python & Java suggests that further learning here is worthwhile.

Compared search activity between 2019 and 2020, and identified the following trends, all of which should help developers and tech leaders invest their time wisely in the months ahead.



Top 25 search terms on online learning platform in 2020.



Top 25 search terms on online learning platform and their year-over-year rate of change in search frequency.

Why you should learn Python?

Python is a rare bird among programming languages. Few languages are prevalent in so many disciplines: web development, data analytics, biology, operations, robotics, graphics, image manipulation, etc. Fewer can boast being almost 30 years old to boot. This combination – a professionally diverse community feeding a platform for nearly three decades – puts Python in a rather magical space.

<Q1> **What's the most important thing happening in Python right now**?

Python **adoption is accelerating** in key industries, due at least in part to the surge of interest in data science and machine learning—spaces where Python has become a staple technology. Looking at patterns of StackOverflow queries across languages, the growth of Python is clearly being demonstrated

<Q2> **What's an adjacent skill/technology that complements Python**?

At present, the most commonly cited reasons to learn Python are its applications in **data science** and **software development**. However, there really isn't a field where you wouldn't benefit from knowing Python. Python is used as an automation platform for several graphics and design applications, it's an omnipresent language in cloud and datacenter operations, it's used routinely in massive biological analytics applications like the Human Genome Project, it's a common platform in robotics programming

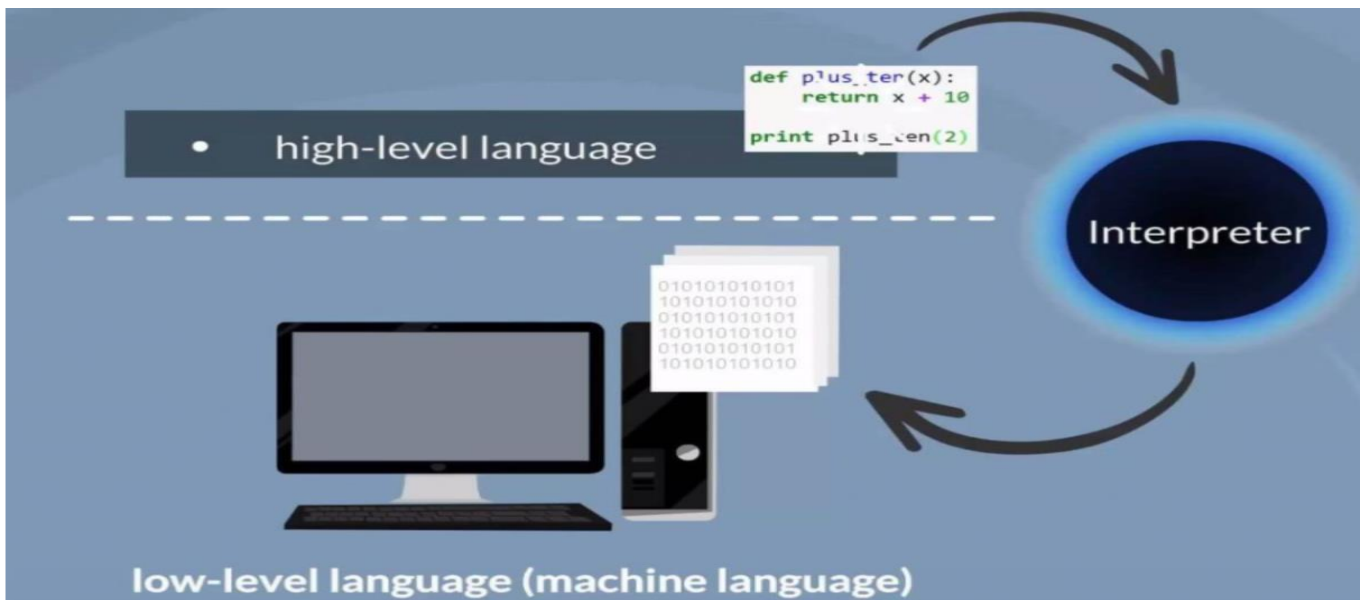
<Q3> **What does the future look like for Python**?

If Python isn't everywhere, it will be soon. One of the biggest indicators I can find at the moment is Microsoft is considering adding support for the language to its flagship Excel office product

Part of the reason that **Python is so successful** is because it's comparatively simple to many other programming languages and platforms. The language syntax is very straightforward; however the **platform built around that language is immense** and can feel very daunting.

The Python Programming Language

- is **interpreted**
  - Interpreted Language vs. Compiled Language … next slide
- is **high-level**
  - Strong abstraction
- is **general-purpose**
  - Used in wide variety of application domains
- is **dynamically typed**
  - Dynamically-typed vs Statically-typed … slide #4
- is **garbage-collected**
  - Makes memory management easier/efficient



Compiled Language vs Interpreted Language

Comparison Chart

Compiled Language	Interpreted Language
The code of compiled languages can be executed directly by the computer's CPU.	A program written in an interpreted language is not compiled, it is interpreted.
The source code must be transformed into machine readable instructions prior to execution.	It does not compile the source code into machine language prior to running the program.
Compiled programs run faster than interpreted programs.	Interpreted programs can be modified while the program is running.
Delivers better performance.	Delivers relatively slower performance.
C, Fortran, and COBOL are languages used to produce compiled programs.	Java and C# are compiled into bytecode, the virtual interpreted language.

Dynamically-Typed vs Statically-Typed

Statically typed programming languages do type checking (i.e. the process of verifying and enforcing the constraints of types) at **compile-time** as opposed to run-time.

Dynamically typed programming languages do type checking at **run-time** as opposed to compile-time.

Advantages/Disadvantages

- Statically typed languages
  - Type checking can be done at compile time
  - Memory layout can be determined at compile time for automatic variables
  - EFFICIENCE
- Dynamically typed languages
  - FLEXIBILITY
  - Example:

```
function max (left, right) {
    if (left < right) return right;
   return left;
}
```

Python – A multi-paradigm Programming Language

- **Object-oriented programming (*)**
  - Fully supported – everything is an object
- **Structured programming**
  - Fully supported
- **Functional programming (*)**
  - Supported
- **Aspect-oriented programming** (metaprogramming, metaobjects)
  - Supported
- **Design-by-Contrast (DbC)**
  - Supported by extensions
- **Logic programming**
  - Supported by extensions

(*) comparison … next slide

PACE UNIVERSITY

Functional Programming vs OOP

Object-Oriented	Functional
Data and the operations upon it are tightly coupled.	Data is only loosely coupled to functions.
Objects hide their implementation of operations from other objects via their interfaces.	Functions hide their implementation, and the language's abstractions speak to functions and the way they are combined or expressed.
The central model for abstraction is the data itself, thus the value of a term isn't always predetermined by the input (stateful approach).	The central model for abstraction is the function, not the data structure, thus the value of a term is always predetermined by the input (stateless approach).
The central activity is composing new objects and extending existing objects by adding new methods to them.	The central activity is writing new functions.

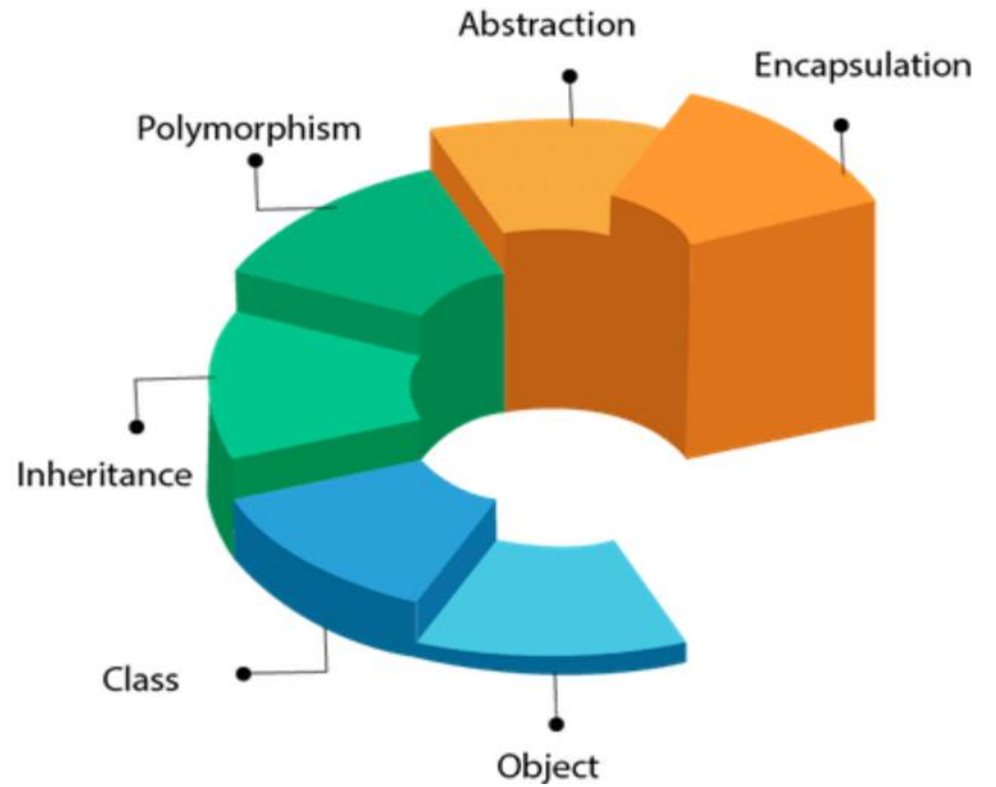
The Python Programming Language

There are two major Python versions, Python 2 and Python 3

Can invoke Python interactively (prompt) or via script (script.py)

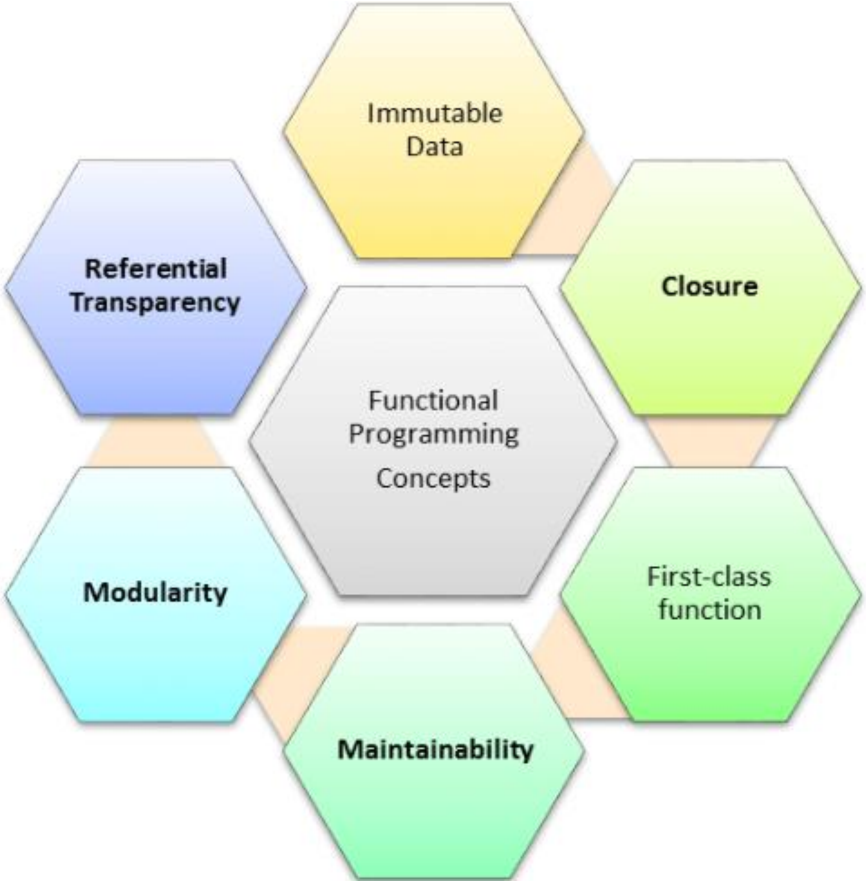
PACE UNIVERSITY

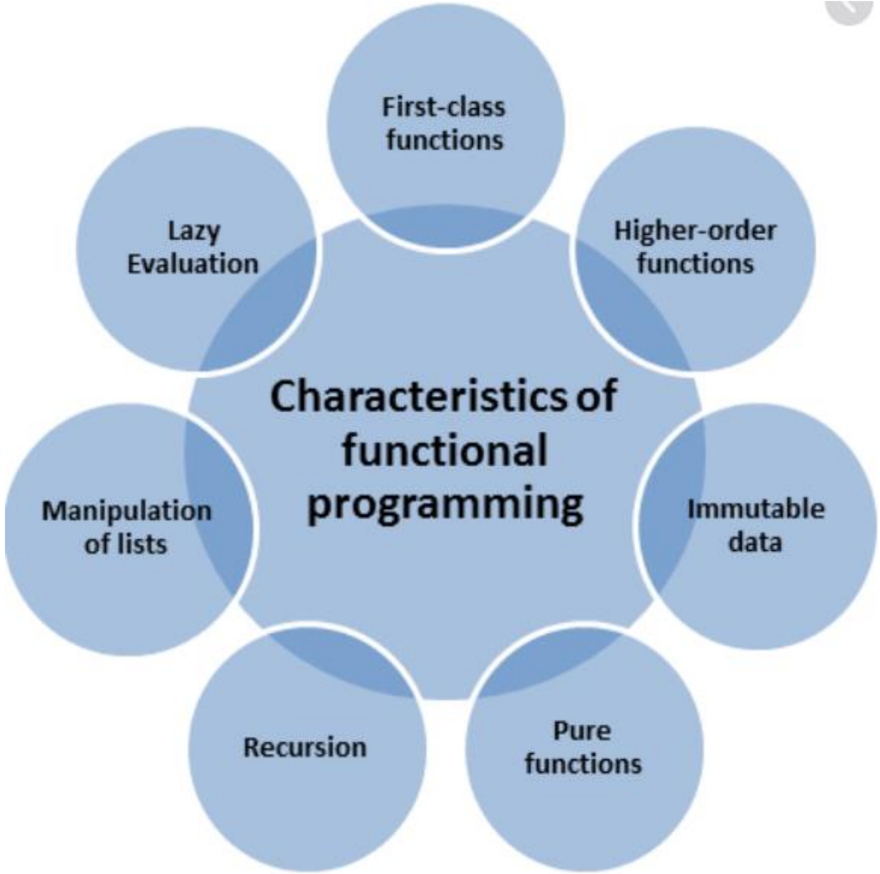
Object-Oriented Programming Language



OO Programming Languages: Java, C++, C#, Ruby, Perl, Python,…
Define the data type of a data structure, and also the types of operations (functions/methods) that can be applied to the data structure.

Functional Programming Language





Functional Programming Languages: Scala, Lisp, Earlang, Haskel, Clojure, Python

Designed to handle symbolic computation and list processing applications. Based on mathematical functions.

Python vs Scala

Comparison Chart

Python	Scala
It a dynamically typed language in which the type checking is done at run-time.	It is a statically typed language in which the type checking is done at compile-time.
It does not support heavyweight process forking so it is not the preferred choice of language for highly concurrent and scalable systems.	It offers multiple asynchronous libraries and reactive cores that help in quick integration of databases in highly scalable systems.
It was originally conceived as an object-oriented language & can be used as a procedural language.	It is the mix of object-oriented and functional programming language.
It is generally easier to learn and use than other programming languages.	It is less difficult to use and learn than Python.

History of Python

- **Born ‘80s**
  - Guido van Rossum
- **Influenced by**
  - Successor to ABC language
- **Name root**
  - Month Python (The Pythons in 1969) keeping it fun to use.
- **Python 2.0**
  - October 2000
- **Python 3.0 (*)**
  - December 2008
- **Python 2.7**
  - EoL: initially set at 2015 then postponed to 2020
- **Python Latest News**
  - Always check PSF (Python Software Foundation – python.org)

PSF: PyPI – The Python Package Index

The **Python Package Index (PyPI)** is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community.

Package authors use PyPI to distribute their software.

https://pypi.org/

PSF: PEPs – Python Enhancement Proposals

A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment.

The PEP should provide a concise technical specification of the feature and a rationale for the feature

https://pypi.org/dev/peps/

Python "One and Only One" Design

Python strives for a simpler, less-cluttered syntax and grammar while giving developers a choice in their coding methodology. In contrast to Perl's "there is more than one way to do it" motto, Python embraces a "**there should be one—and preferably only one—obvious way to do it**" design philosophy.

Python's developers strive to avoid **premature optimization.** (*) When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C, or use PyPy, a just-in-time compiler. Cython is also available, which translates a Python script into C and makes direct C-level API calls into the Python interpreter.

(*) Donald Knuth – The root of all evil.

The Python Standard Library

Python is often described as a "**batteries included**" language due to its comprehensive standard library.

Some of the Python Standard Library modules we use in our books and videos
collections—Additional data structures beyond lists, tuples, dictionaries and sets.
csv—Processing comma-separated value files.
datetime, time—Date and time manipulations.
decimal—Fixed-point and floating-point arithmetic, including monetary calculations.
doctest—Simple unit testing via validation tests and expected results embedded in docstrings.
json—JavaScript Object Notation (JSON) processing for use with web services and NoSQL document databases.
math—Common math constants and operations.
os—Interacting with the operating system.
queue—First-in, first-out data structure.
random—Pseudorandom numbers.
re—Regular expressions for pattern matching.
sqlite3—SQLite relational database access.
statistics—Mathematical statistics functions like mean, median, mode and variance.
string—String processing.
sys—Command-line argument processing; standard input, standard output and standard error streams.
timeit—Performance analysis.

The Pythonic Way

A common name (nomenclature) in the Python community is **pythonic**, which can have a wide range of meanings related to program style.

To say that code is pythonic is to say that it **uses Python idioms well**, that it is natural or shows fluency in the language, that it **conforms with Python's minimalist philosophy and emphasis on <u>readability</u>**.

In contrast, code that is difficult to understand or reads like a rough transcription from another programming language is called ‘unpythonic’.

Users and admirers of Python, especially those considered knowledgeable or experienced, are often referred to as **Pythonistas**

Python --- Learn the Basics

There are two major Python versions, Python 2 and Python 3

Can invoke Python interactively (prompt) or via script (script.py)

- **Indentation for blocks**
- **Variables and Types (Imported, Built-In, User-Defined)**
- **Sequences : Lists [] & Tuples ()**
- **Basic Operators**
- **Text Sequences: Strings -- String Formatting**
- **Basic Sting Operations**
- **Conditions**
- **Loops**
- **Functions**
- **Classes and Objects**
- **Mapping Types: Dictionaries {}**
- **Modules, Packages and Special Methods (__init__, __repr__, __dict__)**

Python's Design Philosophy

REPL (**R**ead **E**valuate **P**rint **L**oop)

The **Zen** of Python (Van Rossum's vision) (PEP 20)

>>> **import this**

(The language's core philosophy is summarized in the document)

Python: Indentation

Python uses **indentation for blocks**, instead of curly braces. Both tabs and spaces are supported, but the **standard indentation** requires standard Python code to use **four spaces**.

For example:

```
x = 1
If x == 1
    # indented four spaces
    print("x is 1.")
```

PACE UNIVERSITY

Variables and Types

Python is completely **object oriented**, and **not "statically typed".** You do not need to declare variables before using them, or declare their type. **Every variable in Python is an object**.

Numeric Type: Build-In Type

Numbers

Python supports two types of numbers - integers and floating point numbers

To define an **integer**, use the following syntax:

```
myint = 7
print(myint)
```

To define a **floating point** number, you may use one of the following notations:

```
myfloat = 7.0
print(myfloat)
myfloat = float(7)
print(myfloat)
```

PACE UNIVERSITY

Variables and Types ... cnt'd.

Strings

Strings are defined either with a single quote or a double quotes.

```
mystring = 'hello'
print(mystring)
mystring = "hello"
print(mystring)
```

Note:

```
mystring = "Don't worry about apostrophes"
print(mystring)
```

There are additional variations on defining strings that make it easier to include things such as carriage returns, backslashes and Unicode characters.

```
one = 1
two = 2
three = one + two
print(three)
hello = "hello"
world = "world"
helloworld = hello + " " + world
print(helloworld)
```

PACE UNIVERSITY

Printing Variables --- The Formatters

What are %s, %r, and %d?

They are "**formatters**" (%r is best for debugging, other formats are for actually displaying variables)

They tell Python to take the variable on the right and put it in to replace the %s with its value.

```
my_name = 'Joe Doe'
my_age = 35 # not a lie
my_height = 74 # inches
my_weight = 180 # lbs
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'Brown'

print "Let's talk about %s." % my_name
print "He's %d inches tall." % my_height
print "He's %d pounds heavy." % my_weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
print "His teeth are usually %s depending on the coffee." % my_teeth

# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (
    my_age, my_height, my_weight, my_age + my_height + my_weight)
```

Python-3 'f-String' Formatting Guide

https://realpython.com/python-f-strings/

Reading from Files --- Writing to Files

Python script to copy one file to another.

```
from sys import argv
from os.path import exists

script, from_file, to_file = argv

print "Copying from %s to %s" % (from_file, to_file)

# we could do these two on one line too, how?
in_file = open(from_file)
indata = in_file.read()

print "The input file is %d bytes long" % len(indata)

print "Does the output file exist? %r" % exists(to_file)
print "Ready, hit RETURN to continue, CTRL- C to abort."
raw_input()

out_file = open(to_file, 'w')
out_file.write(indata)

print "Alright, all done."

out_file.close()
in_file.close()
```

Warning: You should always make sure that an open file is properly closed.

Alternative?

Use 'with' statement

This is **standard I/O** capability by using **core Python**.
See below while we use '**pandas**'!

Input from Keyboard

The input function

If the input function is called, the program flow will be stopped until the user has given an input and has ended the input with the return key.
The text of the optional parameter, i.e. the prompt, will be printed on the screen.

```
>>> user_input = input(">")
>Hello World
>>> user_input
'Hello World'
>>> type(user_input)
<class 'str'>
```

Lists

Python has six **built-in types of sequences**, most common ones are **lists** & **tuples**

Lists are very **similar to arrays**.
They can contain **any type of variable**, and they can contain as many variables as you wish. Can be iterated over in a simple manner.

Here is an example of how to build a list.

```
mylist = []
mylist.append(1)
mylist.append(2)
mylist.append(3)
print(mylist[0]) # prints 1
print(mylist[1]) # prints 2
print(mylist[2]) # prints 3

# prints out 1,2,3
for x in mylist:
    print(x)
```

Definition

A **list** is a data structure in Python that is a **mutable**, or changeable, **ordered sequence** of elements. Each element or value that is inside of a list is called an **item**. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ].

Lists are great to use when you want to work with many related values. They enable you to keep data together that belongs together, condense your code, and perform the same methods and operations on multiple values at once.

Lists … cnt'd

Accessing the whole list

```
mylist = [1,2,3]
print(mylist)

# prints out 1,2,3
```

Append vs Extend

```
x = [1, 2, 3]
x.append([4, 5])
print (x)

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

```
x = [1, 2, 3]
x.extend([4, 5])
print (x)

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

Accessing members of list

```
mylist = [1,2,3]
print(mylist[0])

# prints out 1
```

Accessing members of list

```
mylist = [1,2,3]
print(mylist[1])

# prints out 2
```

Accessing members of list

```
mylist = [1,2,3]
print(mylist[2])

# prints out 3
```

Accessing an index which does not exist generates an exception (error).

```
mylist = [1,2,3]
print(mylist[4])

# prints out
File "<stdin>", line 1,
in <module>indexError:
list index out of range
```

Python uses 0-based indexing

Here is a good blog explaining why:
http://python-history.blogspot.com/2013/10/why-python-uses-0-based-indexing.html

Lists --- Accessing Values

list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5, 6, 7];

print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]

When the above code is executed, it produces the following result

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

Updating Lists

print "Value available at index 2 :

print list[2]

list[2] = 2001;

print "New value available at index 2 : "

print list[2]

When the above code is executed, it produces the following result –

Value available at index 2 :

1997

New value available at index 2 :

2001

```
>>> data = [[1, 2, 3, 4],
...         ['blue', 'green', 'red', 'yellow']]
>>> data
[[1, 2, 3, 4], ['blue', 'green', 'red', 'yellow']]
>>> data.append(['GGOOG', 'AMZN', 'AAPL'])
>>> data
[[1, 2, 3, 4], ['blue', 'green', 'red', 'yellow'], ['GGOOG', 'AMZN', 'AAPL']]
>>> data[0]
[1, 2, 3, 4]
>>> data[0][0]
1
>>> data[1]
['blue', 'green', 'red', 'yellow']
>>> data[1][1]
'green'
>>> data[2]
['GGOOG', 'AMZN', 'AAPL']
>>> data[2][2]
'AAPL'
>>>
```

Python's **list of lists**. (e.g. od 2D & 3D lists)

```
>>> a = [-30, -20, -10, 0, 10, 20, 30]
>>> b = [1, 2, 5]
>>> c = [ a[i] for i in b]
>>> c
[-20, -10, 20]
```

Accessing Python's list elements by **using a known list** and create a new one…

Lists --- Delete Elements (del, remove, pop)

```
list1 = ['physics', 'chemistry', 1997, 2000];

print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

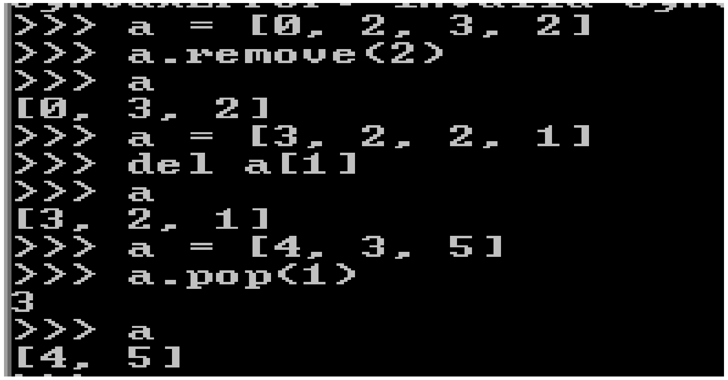
When the above code is executed, it produces following result −

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Remove removes the first matching value, not a specific index

Del removes the item at a specific index

Pop removes the item at a specific index and returns it



Their error modes are different when we are 'index out of range'!

Lists --- Basic 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

Python's List Methods

Here is a list **of built-in methods** that you can use to work with Python Lists.

It is important to keep in mind that lists are mutable —changeable— data types. Unlike strings, which are immutable, when use a method on a list you will be affecting the list itself and not a copy of the list.

list.append()

The method list.append(x) will add an item (x) to the end of a list.

list.insert()

The list.insert(i,x) method takes two arguments, with i being the index position you would like to add an item to, and x being the item itself.

list.extend()

If we want to combine more than one list, we can use the list.extend(L) method, which takes in a second list as its argument.

list.remove()

When we need to remove an item from a list, we'll use the list.remove(x) method which removes the first item in a list whose value is equivalent to x.

list.pop()

We can use the list.pop([i]) method to return the item at the given index position from the list and then remove that item. The square brackets around the i for index tell us that this parameter is optional, so if we don't specify an index, the last item will be returned and removed.

list.index()

When lists start to get long, it becomes more difficult for us to count out our items to determine at what index position a certain value is located. We can use list.index(x), where x is equivalent to an item value, to return the index in the list where that item is located. If there is more than one item with value x, this method will return the first index location.

Python's List Methods ... Cnt'd

list.copy()

When we are working with a list and may want to manipulate it in multiple ways while still having the original list available to us unchanged, we can use list.copy() to make a copy of the list.

list.reverse()

Reverse the order of items in a list by using the list.reverse() method. It is more convenient for us to use reverse alphabetical order rather than traditional alphabetical order. In that case, we need to use the .reverse() method with the fish list to have the list be reversed in place.

list.count()

The list.count(x) method will return the number of times the value x occurs within a specified list. We may want to use this method when we have a long list with a lot of matching values. If we had a larger aquarium, for example, and we had an item for each and every neon tetra that we had, we could use .count() to determine the total number of neon tetras we have at any given time.

list.sort()

We can use the list.sort() method to sort the items in a list.

Just like list.count(), list.sort() can make it more apparent how many of a certain integer value we have, and it can also put an unsorted list of numbers into numeric order.

list.clear()

Can remove all values contained in it by using the list.clear() method.

As a **mutable ordered** sequence of elements, lists are very flexible data structures in Python. Can combine methods with other ways to modify lists to have a full range of tools to use lists effectively in our programs. See **list comprehensions** to create lists based on existing lists.

Tuples --- Read-Only Lists (Immutable Lists)

You define a **tuple** just like you define a list, except you **use parentheses** instead of square brackets.

Once you have a tuple, you can access individual elements just like you can with a list, and you can loop through the tuple with a *for* loop:

```
colors = ('red', 'green', 'blue')
print("The first color is: " + colors[0])

print("\nThe available colors are:")
for color in colors:
    print("- " + color)
```

If you try to add something to a tuple, you will get an error:

```
colors = ('red', 'green', 'blue')
colors.append('purple')
```

Same happens when you try to remove something from a tuple, or modify one of its elements.

Once you define a tuple, you can be confident that its **values will not change**.

Basic Operators

Arithmetic Operators

As other programming languages, addition, subtraction, multiplication, and division operators can be used with numbers.

```
number = 1 + 2 * 3 / 4.0
print(number)
```

Modulo (%) operator

```
remainder = 11 % 3
print(remainder)
```

Power (**) operator

```
squared = 7 ** 2
cubed = 2 ** 3
```

Concatenating strings using the **addition** operator

```
helloworld = "hello" + " " + "world"
print(helloworld)
```

PACE UNIVERSITY

Using Operators with Lists

Lists can be joined with the addition operators:

```
even_numbers = [2,4,6,8]
odd_numbers = [1,3,5,7]
all_numbers = odd_numbers + even_numbers
```

Python also supports multiplying strings to form a string with a repeating sequence:

```
lotsofhellos = "hello" * 10
print(lotsofhellos)
```

Replacing substrings

```
message = "I like cats and dogs, but I'd much rather own a dog."
message = message.replace('dog', 'bird')
print(message)
```

Splitting strings

```
message = "I like cats and dogs, but I'd much rather own a dog."
words = message.split(' ')
print(words)
```

String Formatting

Python uses C-style string formatting to create new, formatted strings

Some basic argument specifiers

```
%s - String (or any object with a string representation, like numbers)
```

```
%d - Integers
```

```
%f - Floating point numbers
```

```
%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.
```

```
%x/%X - Integers in hex representation (lowercase/uppercase)
```

```
# This prints out "Hello, John!“
name = "John“
print("Hello, %s!" % name)
```

```
# This prints out "John is 23 years old."
name = "John“  (check it ‘id’ along with j_name = name…)
age = 23
print("%s is %d years old." % (name, age))
```

```
# This prints out: A list: [1, 2, 3]
mylist = [1,2,3]
print("A list: %s" % mylist)
```

Basic String Operations

```
astring = "Hello world!“
print("single quotes are ' '")
print(len(astring))
```

```
astring = "Hello world!"
print(astring.index("o"))
```

It prints out 4, because the location of the first occurrence of the letter "o" is 4 characters away from the first character. Notice how there are actually two o's in the phrase - this method only recognizes the first.

```
astring = "Hello world!“
print(astring[3:7])
```

```
astring = "Hello world!“
print(astring[3:])
```

```
astring = "Hello world!“
print(astring[:7])
```

```
astring = "Hello world!“
print(astring[-3:])   #3rd character from the end
```

Basic String Operations … cnt'd

A **Python slice** extracts elements, based on a start and stop.

We specify an optional first index, an optional last index, and an optional step.

```
astring = "Hello world!"
print(astring[3:7:2])  #The general form is [start:stop:step].
```

```
#Here is some magic!
astring = "Hello world!"
print(astring[::-1])  #string reversal!
```

```
astring = "AaBbCcDdEe"
print(astring[::2])
print(astring.upper())
print(astring.lower())
```

```
astring = "Hello world!"
print(astring.startswith("Hello"))
print(astring.endswith("abcd"))
afewwords = astring.split(" ")
print(afewwords)
```

PACE UNIVERSITY

Conditions / Boolean Function & Operator

```
x = 2
print(x == 2) # prints out True
print(x == 3) # prints out False
print(x != 3) # prints out True
print(x < 3) # prints out True
```

Note
One (1) equal sign is the **Assignment operator**.
Two (2) equal signs is the **Comparison (Boolean) operator**.

Boolean operators

```
name = "John"
age = 23
if name == "John" and age == 23:
    print("Your name is John, and you are also 23 years old."
)
if name == "John" or name == "Rick":
    print("Your name is either John or Rick.")
```

There is exist a **bool data type** (which inherits from int and has only two values: True and False).
But also Python has the boolean-able concept for every object, which is used when **function bool([x])** is called.

https://docs.python.org/3/library/functions.html#bool

```
name = "John“  #The “in” operator
if name in ["John", "Rick"]:
    print("Your name is either John or Rick.")
```

Note
Unlike Java where you would declare
boolean flag = True
in Python you can just declare myFlag = True

Python would interpret this as a boolean variable

Conditions … cnt'd

Python's "if" statement using code blocks:

```
if <statement is="" true="">:
    <do something="">
    ....
    ....
elif <another statement="" is="" true="">: # else if
    <do something="" else="">
    ....
    ....
else:
    <do another="" thing="">
    ....
    ....
</do></do></another></do></statement>
```

```
x = 2
if x == 2:
    print("x equals two!")
else:
    print("x does not equal to two.")
```

There can be **zero or more elif** parts, and the **else** part is **optional**.
The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation.
An **if … elif … elif … sequence** is a substitute for the switch or case statements found in other languages.

It's very common in C: hacking 'empty if statement' like:

```
if(mostlyhappencondition)
    ;#empty statement
else{
    dosomething;
}
```

Here is the equivalent in Python: the **pass** statement

```
if mostlyhappencondition:
    pass
else:
    do_something()
```

Write **inline if statement** for print:

```
>>> a = 100
>>> b = True
>>> print(a if b else "other number")
100
>>> b = False
>>> print(a if b else "other number")
other number
```

Conditions … cnt'd

Python's "is" operator:

Unlike the double equals operator "==", "is" operator doesn't match the values of the variables, but the instances themselves.

```
x = [1,2,3]
y = [1,2,3]
print(x == y) # Prints out True
print(x is y) # Prints out False
```

The "not" operator:

Using "not" before a boolean expression inverts it:

```
print(not False) # Prints out True
print((not False) == (False)) # Prints out False
```

Conditional Expressions (aka Ternary Operator)

```
conditional_expression ::=  or_test ["if" or_test "else" expression]
expression             ::=  conditional_expression | lambda_expr
expression_nocond      ::=  or_test | lambda_expr_nocond
```

Lambda expressions (sometimes called lambda forms) are used to create **anonymous functions**

```
>>> a = 1
>>> b = 2
>>> 1 if a > b else -1
-1
>>> 1 if a > b else -1 if a < b else 0
-1
```

```
lambda_expr        ::=  "lambda" [parameter_list]: expression
lambda_expr_nocond ::=  "lambda" [parameter_list]: expression_nocond
```

Loops

The "for" loop: For loops iterate over a given sequence

```
primes = [2, 3, 5, 7]
for prime in primes:
    print(prime)
```

range function returns a new list with numbers of that specified range, whereas **xrange** returns an iterator, which is more efficient. range function is <u>zero</u> based.

```
# Prints out the numbers 0,1,2,3,4
for x in range(5):
    print(x)

# Prints out 3,4,5
for x in range(3, 6):
    print(x)

# Prints out 3,5,7
for x in range(3, 8, 2):
    print(x)
```

Python's **range()** Function

The range() function has two sets of parameters, as follows:

range(stop)

stop: Number of integers (whole numbers) to generate, starting from zero. eg. range(3) == [0, 1, 2].

range([start], stop[, step])

start: Starting number of the sequence.
stop: Generate numbers up to, but not including this number.
step: Difference between each number in the sequence.

Note that:
All parameters must be integers.
All parameters can be positive or negative.
range() (and Python in general) is 0-index based

Loops … cnt'd

The "while" loop: repeat as long as a certain boolean condition is met

```
# Prints out 0,1,2,3,4
count = 0
while count < 5:
    print(count)
    count += 1  # This is the same as count = count + 1
```

"**break**" and "**continue**" statements

break is used to exit a for loop or a while loop, whereas continue is used to skip the current block, and return to the "for" or "while" statement.

```
# Prints out 0,1,2,3,4
count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break
#Prints out only odd numbers - 1,3,5,7,9
for x in range(10):
    # Check if x is even
    if x % 2 == 0:
        continue
    print(x)
```

Loops … cnt'd

Can we use "else" clause for loops?

Unlike languages like C,C++.. we can use **else for loops**.

When the loop condition of "for" or "while" statement fails then code part in "else" is executed. If break statement is executed inside for loop then the "else" part is skipped.

Note that "else" part is executed even if there is a continue statement.

```
# Prints out 0,1,2,3,4 and then it prints "count value reached 5"
count=0
while(count<5):
    print(count)
    count +=1
else:
    print("count value reached %d" %(count))
# Prints out 1,2,3,4
for i in range(1, 10):
    if(i%5==0):
        break
    print(i)
else:
    print("this is not printed because for loop is terminated because of break but not due to fail in condition")
```

Functions – Docstring Conventions (PEP-257)

Functions are a convenient way to **divide your code into useful blocks**, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code.

How do you write function in Python?

Python makes use of blocks. A block is an area of code written in the format of:

```
block_head:
    1st block line
    2nd block line
    ...
```

```
def my_function():
    """This is my function…"""
    print("Hello From My Function!")
```

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**.

All **modules** should normally have docstrings, and all **functions** and **classes** exported by a module should also have docstrings. **Public methods** (including the __init__ constructor) should also have docstrings.
A **package** may be documented in the module docstring of the __init__.py file in the package directory.
Read more: https://www.python.org/dev/peps/pep-0257/

Functions may also receive **arguments** (variables passed from the caller to the function).

```
def my_function_with_args(username, greeting):
    "This is my function…"
    print("Hello, %s , From My Function!, I wish you %s"%(username, greeting))
```

Functions … cnt'd

Functions may **return a value** to the caller, using the keyword- '**return**' .

```
def sum_two_numbers(a, b):
    return a + b
```

How do you call functions in Python?

Write the function's name followed by (), placing any required arguments within the brackets.

```
# Define our 3 functions
def my_function():
    print("Hello From My Function!")
def my_function_with_args(username, greeting):
    print("Hello, %s , From My Function!, I wish you %s"%(username, greeting))
def sum_two_numbers(a, b):
    return a + b

# print(a simple greeting)
my_function()
#prints - "Hello, John Doe, From My Function!, I wish you a great year!"
my_function_with_args("John Doe", "a great year!")
# after this line x will hold the value 3!
x = sum_two_numbers(1,2)
```

Function w/ default arguments

```
def foo(a, b, x=3, y=2):
    return (a+b)/(x+y)
```

```
>>> def foo(a, b, x=3, y=2):
...     return(a+b)/(x+y)
...
>>> foo(5,0)
1.0
>>> foo(10,2,y=3)
2.0
>>> foo(b=4, x=8, a=1)
0.5
>>>
```

Function Accepting zero (0) or more arguments

Sometimes you want to define a function with any number of arguments.

```
# Note the asterisk. That's the magic part
def takes_any_args(*args):
    print("Types of args: " + str(type(args)))
    print("Values of args: " + str(args))
```

```
>>> takes_any_args(1)
Types of args: <class 'tuple'>
Values of args: (1,)
>>> takes_any_args('x','y','z')
Types of args: <class 'tuple'>
Values of args: ('x', 'y', 'z')
>>> takes_any_args()
Types of args: <class 'tuple'>
Values of args: ()
>>>
```

Very Important: Within the function ***args** is a **tuple**

Single Argument vs. *args

```
def takes_any_args(*args):
    print("Types of args: " + str(type(args)))
    print("Values of args: " + str(args))

def takes_a_list(items):
    print("Types of items: " + str(type(items)))
    print("Values of items: " + str(items))
```

```
>>> data = ["x", "y", "z"]
>>> takes_any_args(data)
Types of args: <class 'tuple'>
Values of args: (['x', 'y', 'z'],)
>>> takes_a_list(data)
Type of items: <class 'list'>
Vale of items: ['x', 'y', 'z']
```

Test: Call functions w/out params…

Note: **Tuple with one element/member.**
<Q> How to distinguish it from a string?

```
>>> type( ('a') )
<type 'str'>

>>> type( ('a',) )
<type 'tuple'>
```

```
>>> a = [('a'), ('b'), ('c', 'd')]
>>> a
['a', 'b', ('c', 'd')]
>>>
>>> for elem in a:
...     print type(elem)
...
<type 'str'>
<type 'str'>
<type 'tuple'>
```

*args improves Function's arguments readability

By convention, the **tuple argument's default name** is args, but it doesn't have to be:

```
def read_files(*paths):
    data = ""
    for path in paths:
        with open(path) as handle:
            data += handle.read()
    return data

# ch1.txt has text of Chapter 1; ch2.txt for Ch. 2, etc.
story = read_files("ch1.txt", "ch2.txt", "ch3.txt", "ch4.txt")
```

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args("red", "blue", "green")
```

```
red
blue
green
```

Keyword arguments can **NOT** be captured by the *args idiom:

```
def print_args(*args):
    for arg in args:
        print(arg)
```

```
>>> print_args(a=4, b=7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_args() got an unexpected keyword argument 'a'
```

<Q> What do we do?

<A> kwargs: **Variable Keyword Arguments**

kwargs: Variable Keyword Arguments

For keyword arguments, use the ****kwargs** syntax. **kwargs** is a **python dictionary**.

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))
```

```
>>> print_kwargs(hero="Homer", antihero="Bart", genius="Lisa")
hero -> Homer
genius -> Lisa
antihero -> Bart
```

Combine args with kwargs

```
def print_all(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))
```

A defined function can use either *args or **kwargs, or both

Positional arguments + kwargs

```
def add_to_dict(stuff, **kwargs):
    for key, value in kwargs.items():
        # Do not overwrite existing values.
        if key not in stuff:
            stuff[key] = value
    return stuff
```

```
>>> add_to_dict({})
{}
>>> add_to_dict({}, foo=42)
{'foo': 42}
>>> add_to_dict({"foo": 42}, bar=21)
{'bar': 21, 'foo': 42}
>>> add_to_dict({"foo": 42}, foo=21)
{'foo': 42}
```

Here is a problem (Functions w/ incompatible types)

Library A defines this function:

```
def order_book(title, author, isbn):
    """
    Place an order for a book.
    """
    print("Ordering '{}' by {} ({})".format(title, author, isbn))
    # ...
```

Library B defines this function:

```
def get_required_textbook(class_id):
    """
    Returns a tuple (title, author, ISBN)
    """
    # ...
```

<Q> How do we combine these 2 libraries?

See next slide

Python's Argument Unpacking

Here is a way of doing this.

But it is tedious and error-prone.

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

<Q> Is there a better way?

```
>>> def normal_function(a, b, c):
...     print("a: {} b: {} c: {}".format(a,b,c))
...
```

<A> Python provides **Arg Unpacking**

```
>>> numbers = (7, 5, 3)
>>> normal_function(*numbers)
a: 7 b: 5 c: 3
>>> # Exactly equivalent to:
... normal_function(numbers[0], numbers[1], numbers[2])
a: 7 b: 5 c: 3
```

Note: normal_funtction is just a regular function! This is called **argument unpacking.**

Argument Unpacking

Given these:

```
one_args = [ 42 ]
two_args = (7, 10)
three_args = [1, 2, 3]

def f(n): return n / 2
def g(a, b): return a + b
def h(x, y, z): return x * y * z
```

All these pairs are equivalent:

```
f(*one_args)
f(one_args[0])

g(*two_args)
g(two_args[0], two_args[1])

h(*three_args)
h(three_args[0], three_args[1], three_args[2])
```

So instead of this....

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

We can do this:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(*book_info)
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

Keyword Unpacking

Just like with *args, double-star works the other way too. We can take a regular function, and pass it a dictionary using two asterisks:

```
def normal_function(a, b, c):
    print("a: {} b: {} c: {}".format(a,b,c))
```

```
>>> numbers = {"a": 7, "b": 5, "c": 3}
>>> normal_function(**numbers)
a: 7 b: 5 c: 3
>>> # Exactly equivalent to:
... normal_function(a=numbers["a"], b=numbers["b"], c=numbers["c"])
a: 7 b: 5 c: 3
```

Matching Keys

Caution

Keys of the dictionary must match up with how the function is declared

```
>>> bad_numbers = {"a": 7, "b": 5, "z": 3}
>>> normal_function(**bad_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: normal_function() got an unexpected keyword argument 'z'
```

Calling Both

You can call a function using both – and **both** will be **unpacked**

```
>>> def addup(a, b, c=1, d=2, e=3):
...     return a + b + c + d + e
...
>>> nums = (3, 4)
>>> extras = {"d": 5, "e": 2}
>>> addup(*nums, **extras)
15
```

PACE UNIVERSITY

Variable Arguments vs. Argument Unpacking

Python uses * and ** for two very different things!!

Variable arguments <<< === >>> When **Defining a function**

Argument Unpacking <<< === >>> When **Calling a function**

Two Different Things

These look similar in code, but **they are completely different things**.

The Function Object

In Python, **everything** is an object. Including functions.

```
>>> def f(n): return n+2
...
>>> id(f)
4314937816
>>> g = f
>>> print(g(3))
5
>>> id(g)
4314937816
```

Function **id** returns a number that is unique to an object

When in Python kernel, type the name of a function then press <tab> to see all the classes of the object

```
>>> takes_any_args.
takes_any_args.__annotations__   takes_any_args.__hash__(
takes_any_args.__call__(         takes_any_args.__init__(
takes_any_args.__class__(        takes_any_args.__kwdefaults__
takes_any_args.__closure__       takes_any_args.__le__(
takes_any_args.__code__          takes_any_args.__lt__(
takes_any_args.__defaults__      takes_any_args.__module__
takes_any_args.__delattr__(      takes_any_args.__name__
takes_any_args.__dict__          takes_any_args.__ne__(
takes_any_args.__dir__(          takes_any_args.__new__(
takes_any_args.__doc__           takes_any_args.__qualname__
takes_any_args.__eq__(           takes_any_args.__reduce__(
takes_any_args.__format__(       takes_any_args.__reduce_ex__(
takes_any_args.__ge__(           takes_any_args.__repr__(
takes_any_args.__get__(          takes_any_args.__setattr__(
takes_any_args.__getattribute__( takes_any_args.__sizeof__(
takes_any_args.__globals__       takes_any_args.__str__(
takes_any_args.__gt__(           takes_any_args.__subclasshook__(
>>> takes_any_args.__class__
<class 'function'>
```

Variable Positional Arguments vs. Keyword Arguments

Python uses * and ** for two very different things!!

Variable arguments <<< === >>> When **Defining a function**

Argument Unpacking <<< === >>> When **Calling a function**

Must know and remember:

***args** = **list** of arguments - as positional arguments (tuple)

****kwargs** = **dictionary** – its keys become separate keyword arguments and the values become values of these arguments.

Usage

Use ***args** when you're not sure how many arguments might be passed to your function, i.e. it allows you pass an arbitrary number of arguments to your function.

When ****kwargs** is used, it allows you to handle named arguments that you have not defined in advance.

Max Functions

See the below ‘max’ functions, given the following inputs: nums = [“12”, “7”, “30”, “14”, “3”]

input: nums = [“12”, “7”, “30”, “14”, “3”]

```
>>> def max_by_int_value(items):
...     # For simplicity, assume len(items) > 0
...     biggest = items[0]
...     for item in items[1:]:
...         if int(item) > int(biggest):
...             biggest = item
...     return biggest
...
>>> max_by_int_value(nums)
'30'
```

input: integers = [3, -2, 7, -1, -20]

```
>>> def max_by_abs(items):
...     biggest = items[0]
...     for item in items[1:]:
...         if abs(item) > abs(biggest):
...             biggest = item
...     return biggest
...
>>> max_by_abs(integers)
-20
```

input:
student_joe = {‘gpa’: 3.7, major’: ‘physics’,
‘name’: ‘Joe Smith’}
student_jane = {‘gpa’: 3.8, major’: ‘chemistry’,
‘name’: ‘Jane Jones’}
student_zoe = {‘gpa’: 3.4, major’: ‘literature’,
‘name’: ‘Zoe Grimwald’}

```
>>> def max_by_gpa(items):
...     biggest = items[0]
...     for item in items[1:]:
...         if item["gpa"] > biggest["gpa"]:
...             biggest = item
...     return biggest
...
>>> max_by_gpa(students)
{'name': 'Jane Jones', 'gpa': 3.8, 'major': 'chemistry'}
```

<Q> What is the difference in these three functions?
<A> Just one line of code is different between *max_by_int_value*, *max_by_abs*, and *max_by_gpa*

```
# for max_by_int_value
if int(item) > int(biggest):

# for max_by_abs
if abs(item) > abs(biggest):

# for max_by_gpa
if item["gpa"] > biggest["gpa"]:
```

Other than that, the max functions are **identical.**

<Q> Could we combine all these functions to one?
<A> See next slide….

Comparison Key Function -- Function as an Argument

Let's define a ***key*** function for each of them, which extracts the relevant value:

```
# for max_by_int_value
int

# for max_by_abs
abs

# for max_by_gpa
def get_gpa(student): return student["gpa"]
```

Max Function by Key

Let's define a generic ***max*** function

```
>>> def max_by_key(items, key):
...     biggest = items[0]
...     for item in items[1:]:
...         if key(item) > key(biggest):
...             biggest = item
...     return biggest
...
```

```
>>> # Old way:
... max_by_int_value(nums)
'30'
>>> # New way:
... max_by_key(nums, int)
'30'
>>> # Old way:
... max_by_abs(integers)
-20
>>> # New way:
... max_by_key(integers, abs)
-20
```

Using the Key Function

This is the important line:

```
# key is actually int, abs, etc.
if key(item) > key(biggest):
```

Important:

Never invoke the key function yourself, pass the function object to max_by_key, which invokes it for you.

Note: For the student GPA , there is not built-in, we have to create our own:

```
>>> # Old way:
... max_by_gpa(students)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}
```

```
>>> # New way:
... def get_gpa(who):
...     return who["gpa"]
...
>>> max_by_key(students, get_gpa)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}
```

Passing Functions to Functions

Similar to the *max_by_key* function we defined, even the built-in ones work too:

```
>>> nums = ["12", "7", "30", "14", "3"]
>>> max(nums)
'7'
>>> max(nums, key=int)
'30'
```

```
>>> nums = ["12", "7", "30", "14", "3"]
>>> max(nums)
'7'
>>> max(nums, key=int)
'30'
>>>
```

It also works for ***max***, ***min***, and ***sorted*** built-in functions.

```
>>> # Default behavior...
... min(nums)
'12'
>>> sorted(nums)
['12', '14', '3', '30', '7']
>>>
>>> # And with a key function:
... min(nums, key=int)
'3'
>>> sorted(nums, key=int)
['3', '7', '12', '14', '30']
```

Note: Many algorithms in ML & DL can be expressed using min, max, or sorted, along with an appropriate **key function**!!!

Warning: Use "key=“

Make sure to use the keyword **'key=‘** in your function argument list:

```
>>> nums = ["12", "7", "30", "14", "3"]
>>> # This works...
... max(nums, key=int)
'30'
```

```
>>> # And this does not.
... max(nums, int)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unorderable types: type() > list()
```

Functions Returning Functions

Note: itemgetter is a function that create and returns a function.

The following two key functions are completely equivalent:

```
# What we did above:
def get_gpa(who):
    return who["gpa"]

# Using itemgetter instead:
from operator import itemgetter
get_gpa = itemgetter("gpa")
```

← works same as this →

```
def itemgetter(dict_key):
    def key_func(mapping):
        return mapping[dict_key]
    return key_func
```

Getting attributes

operator.itemgetter does something similar for *classes*

```
>>> class Student:
...     def __init__(self, name, major, gpa):
...         self.name = name
...         self.major = major
...         self.gpa = gpa
...     def __repr__(self):
...         return "{}: {}".format(self.name, self.gpa)
...
>>> student_objs = [
...     Student("Joe Smith", "physics", 3.7),
...     Student("Jane Jones", "chemistry", 3.8),
...     Student("Zoe Grimwald", "literature", 3.4),
...     ]
>>> from operator import attrgetter
>>> sorted(student_objs, key=attrgetter("gpa"))
[Zoe Grimwald: 3.4, Joe Smith: 3.7, Jane Jones: 3.8]
```

The Operator Module
Standard operators as functions

Operation	Syntax	Function
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`

The **operator module** exports a set of efficient functions corresponding to the intrinsic operators of Python.
E.g., operator.add(x, y) is equivalent to the expression x+y.
Find a complete list by visiting the url below:
https://docs.python.org/3/library/operator.htm

Python Decorators (PEP 318)

A **decorator** is a way to **add behavior** around a function or method.

The general syntax is:

```
@somedecorator
def some_function(x, y, z):
    # ...
```

Writing decorators is very challenging

What it lets you do:

- Add rich features to functions & classes
- Encapsulate code reuse patterns (otherwise impossible)
- Effectively extend Python syntax
- Build easily reusable frameworks
- Untangle intertwined concerns in your code

```
>>> class Person:
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...
...     @property
...     def full_name(self):
...         return self.first_name + " " + self.last_name
...
>>> person = Person("John", "Smith")
>>> print(person.full_name)
John Smith
```

It is just a function

A decorator is **just a function**.
It is a function that takes exactly one argument, which is a function object.
And it returns a different function

Terminology

```
@some_decorator
def some_function(arg):
    # blah blah
```

- ***decorator*** – What comes after the @. It is a function
- ***bare function*** – the one defined on the next line. The function being decorated.
- The result of decorating a function is the ***decorated function***. It's what you actually call in your code

Decorators – Remember One Thing

A **decorator is just a normal**, boring **function**!

It happens to be a function that **takes exactly one argument**, which is itself a function.

And when called, the decorator **returns a different function**.

Example: **Logging decorator**

```
def printlog(func):
    def wrapper(arg):
        print("CALLING: " + func.__name__)
        return func(arg)
    return wrapper

@printlog
def f(n):
    return n+2

# Same as:
def f(n):
    return n+2
f = printlog(f)
```

```
>>> print(f(3))
CALLING: f
5
```

Masking

```
def check_id(func):
    def wrapper(arg):
        print("ID of func: {}".format(id(func)))
        return func(arg)
    print("ID of wrapper: {}".format(id(wrapper)))
    return wrapper
```

```
>>> @check_id
... def f(x): return x * 3
ID of wrapper: 4329698984
>>>
>>> f(2)
ID of func: 4329698576
6
>>> id(f)
4329698984
```

Read more here: https://wiki.python.org/moin/PythonDecorators

Classes and Objects

Objects are an **encapsulation** of **variables** and **functions** into a single entity. Objects get their variables and functions from **classes.**

Classes are essentially a **template** to create your objects.

```
class MyClass:
    variable = "blah"

    def function(self):  # ‘self’ is just a place holder
        print("This is a message inside the class.")
```

A **Type** is a version of a **Class**.
A Class is the definition of that Type.
An **instance** is what is created once you use (**instantiate**) a Class

Then to assign the above class(template) to an object you would do the following:

```
myobjectx = MyClass()
```

#"myobjectx" holds an object of class "MyClass" that contains the variable & the function defined within the class called "MyClass“

Accessing Object Variables/Functions

```
myobjectx.variable
print(myobjectx.variable)
myobjectx.function()
```

Convention
The **name of a Python Class** starts with a **Capital** letter.

#Can create multiple different objects that are of the same class (have same variables & functions defined). However, each object contains independent copies of the variables defined in the class.

Python's Special Methods

Some **special method names** that start with double underscore '__' and end with double underscore '__' in Python are very useful (**dunder methods**):

__init__ : called whenever a class instance is created (constructed)
To control the actual creation process, use the **__new__()** method.
__repr__ : how to string represent the object
__dict__ : a dictionary support the namespace within the class
__str__ : this method is used when print(x) is called
__call__ : instances behave like functions, be called like a function

You Want...	So You Write...	And Python Calls...
to initialize an instance	`x = MyClass()`	`x.__init__()`
the "official" representation as a string	`repr(x)`	`x.__repr__()`
the "informal" value as a string	`str(x)`	`x.__str__()`
the "informal" value as a byte array	`bytes(x)`	`x.__bytes__()`
the value as a formatted string	`format(x, format_spec)`	`x.__format__(format_spec)`

```
for x in seq:
    print(x)
```

For this loop Python 3 will call **seq.__iter__()** to create an **iterator**, then call the **__next__() method** on that iterator to get each value of x. When __next__() method raises a StopIteration exception, for loop ends gracefully

PACE UNIVERSITY

Python Classes – Examples (Simple)

Class (simple)

class Greeter(*object*):

def hello(*self*):
    print("Hello")

def goodbye(self):
    print("Goodbye")

\>>> g = Greeter()
\>>> g.hello()
\>>> g.goodbye()

\>>> g2 = Greeter()
\>>> g2.hello()
\>>> g2.goodbye()

```
>>> g = Greeter()
>>> g.hello()
Hello
>>> g.goodbye()
Goodbye
>>> g2 = Greeter()
>>> g2.hello()
Hello
>>> g2.goodbye()
Goodbye
```

Class (with constructor)

class Greeter(*object*):

def __init__(*self*, name):
    self.name = name

def hello(*self*):
    print("Hello" + " " + self.name)

def goodbye(self):
    print("Goodbye"+ " " +
self.name)

\>>> g = Greeter("Alice")
\>>> g.hello()
\>>> g.goodbye()

\>>> g2 = Greeter("Bob")
\>>> g2.hello()
\>>> g2.goodbye()

```
>>> class Greeter(object):
...
...     def __init__(self, name):
...         self.name = name
...
...     def hello(self):
...         print("Hello" + " " + self.name)
...
...     def goodbye(self):
...         print("Goodbye" + " " + self.name)
...
>>> g = Greeter("Alice")
>>> g.hello()
Hello Alice
>>> g.goodbye()
Goodbye Alice
>>> g2 = Greeter("Bob")
>>> g2.hello()
Hello Bob
>>> g2.goodbye()
Goodbye Bob
```

Python Classes – Examples (A bit complex)

Class (standard dice)

```
import random

class Die(object):
    def roll(self):
        return random.randint(1, 6)
```

```
>>> d = Die()
>>> print(d.roll())
>>> print(d.roll())
>>> print(d.roll())

>>> d2 = Die()
>>> print(d2.roll())
```

```
>>> import random
>>> class Die(object):
...
...     def roll(self):
...         return random.randint(1, 6)
...
>>> d = Die()
>>> print(d.roll())
5
>>> print(d.roll())
4
>>> print(d.roll())
6
>>>
>>> d2 = Die()
>>> print(d2.roll())
5
>>> print(d2.roll())
1
```

Class (Any size dice)

```
import random

class Die(object):
    def __init__(self, sides):
        self.sides = sides

    def roll(self):
        return random.randint(1,
self.sides)
```

```
>>> class Die(object):
...
...     def __init__(self, sides):
...         self.sides = sides
...
...     def roll(self):
...         return random.randint(1, self.sides)
...
```

```
>>> print("D6 rolls:")
>>> d = Die(6)
>>> print(d.roll())
>>> print(d.roll())
>>> print(d.roll())
```

```
>>> print("D6 rolls:")
D6 rolls:
>>> d = Die(6)
>>> print(d.roll())
1
>>> print(d.roll())
6
>>> print(d.roll())
3
>>>
```

```
>>> print("D20 rolls:")
>>> d2 = Die(20)
>>> print(d2.roll())
>>> print(d2.roll())
>>> print(d2.roll())
```

```
>>> print("D20 rolls:")
D20 rolls:
>>> d2 = Die(20)
>>> print(d2.roll())
5
>>> print(d2.roll())
19
>>> print(d2.roll())
3
```

Python H/W -- Assignment

Simulate the activity of a casino dealer who performs the following 2 actions:

1) **Shuffle** the cards, and

2) **Return** a random card to you

Recall, a card has:

A) 4 **suits**: Clubs, Diamonds, Hearts, and Spades

B) 13 **ranks** = 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace

Hint:

Create a Python class, call it, Deck.

- Build the full deck with all 52 cards
- Randomly select a card off the deck

Dictionaries

A **dictionary** is a data type similar to arrays, but works with **keys** and **values** instead of indexes. Each value stored in a dictionary can be accessed using a key, which is any type of object (a string, a number, a list, etc.) instead of using its index to address it.

```
phonebook = {}
phonebook["John"] = 987654321
phonebook["Jack"] = 123456789
phonebook["Jill"] = 246801359
print(phonebook)
```

Alternatively, a dictionary can be initialized with the same values

```
phonebook = {
    "John" : 938477566,
    "Jack" : 938377264,
    "Jill" : 947662781
}
print(phonebook)
```

Notes on Dictionaries

- Dicts are not sorted. So they don't retain order.
- Keys must be unique
- Dicts point to data values, so they retain changes
- The types used for keys and values do not need to be same even inside same dict

Dictionaries … cnt'd

Iterating over dictionaries

Dictionaries can be iterated over, just like a list. However, a dictionary, unlike a list, does not keep the order of the values stored in it. Here is how to iterate over key value pairs:

```
phonebook = {"John" : 938477566,"Jack" : 938377264,"Jill" : 947662781}
for name, number in phonebook.items():
    print("Phone number of %s is %d" % (name, number))
```

Removing a Value

```
phonebook = {
   "John" : 938477566,
   "Jack" : 938377264,
   "Jill" : 947662781}
del phonebook["John"]
print(phonebook)
```

Or

```
phonebook = {
   "John" : 938477566,
   "Jack" : 938377264,
   "Jill" : 947662781}
phonebook.pop("John")
print(phonebook)
```

Modules and Packages

Modules are Python files with .py extension, which implement a set of functions.
Modules are imported from other modules using the **import** command.

To import a module, we use the **import** command.
Check out the full list of built-in modules in the **Python standard library**
https://docs.python.org/2/library/ or **https://docs.python.org/3/library/**

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module imports the same module, it won't be loaded twice but once only - local variables inside the module act as a "singleton"- they are **initialized once**.

E.g., to import the module **urllib**, which enables us to create read data from URLs:

```
# import the library
import urllib

# use it
urllib.urlopen(...)
```

PACE UNIVERSITY

Modules and Packages … cnt'd

Exploring built-in modules

Two very important functions come in handy when exploring modules in Python - the **dir** and **help** functions.

You can look for which functions are implemented in each module by using the dir function

```
>>> import urllib
>>> dir(urllib)['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLopener', '__all__',
'__builtins__', '__doc__', '__file__', '__name__', '__package__', '__version__', '_ftperrors', '_get_proxies',
'_get_proxy_settings', '_have_ssl', '_hexdig', '_hextochr', '_hostprog', '_is_unicode', '_localhost', '_noheaders',
'_nportprog', '_passwdprog', '_portprog', '_queryprog', '_safe_map', '_safe_quoters', '_tagprog', '_thishost',
'_typeprog', '_urlopener', '_userprog', '_valueprog', 'addbase', 'addclosehook', 'addinfo', 'addinfourl',
'always_safe', 'basejoin', 'c', 'ftpcache', 'ftperrors', 'ftpwrapper', 'getproxies', 'getproxies_environment',
'getproxies_macosx_sysconf', 'i', 'localhost', 'main', 'noheaders', 'os', 'pathname2url', 'proxy_bypass',
'proxy_bypass_environment', 'proxy_bypass_macosx_sysconf', 'quote', 'quote_plus', 'reporthook', 'socket',
'splitattr', 'splithost', 'splitnport', 'splitpasswd', 'splitport', 'splitquery', 'splittag', 'splittype', 'splituser', 'splitvalue',
'ssl', 'string', 'sys', 'test', 'test1', 'thishost', 'time', 'toBytes', 'unquote', 'unquote_plus', 'unwrap', 'url2pathname',
'urlcleanup', 'urlencode', 'urlopen', 'urlretrieve']
```

Then you can read about it more using the help function, inside the Python interpreter:

```
help(urllib.urlopen)
```

Modules and Packages … cnt'd

>>> Writing modules <<<

To create a module of your own, simply create a new **.py file** with the module name, and then import it using the Python file name (without the .py extension) using the **import command**.

>>> Writing packages <<<

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist. **Each package in Python is a directory** which **MUST** contain a special file called **__init__.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the *__init__.py* file inside the foo directory.

To use the module bar, we can import it in two ways:

```
import foo.bar
```

Or

```
from foo import bar
```

Python Scopes & Namespaces

In the first method, import foo.bar, we must use the foo **prefix** whenever we access the module bar. (i.e. import foo.bar)

In the second method, from foo import bar, we don't, because we import the module to our **module's namespace**. (from foo import bar)

The __init__.py file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the __all__ variable, like so:

__init__.py:

```
__all__ = ["bar"]
```

Read more about **Namespaces**, here: https://docs.python.org/3/tutorial/classes.html

Exercise (Assignment)

Print an alphabetically sorted list of all functions in the **re** module, which contain the word **find**.

```
import re

# Your code goes here
```

Hint
- Initialize an empty list
- Each ‘member’ in ‘re’ module that has ‘find’ add it into the list
- Print the list (sorted)

Data Science Basics

Fundamental Libraries for Scientific Computing

- **Jupyter Notebook** (aka IPython Notebook)
- **Numpy**
- **Pandas**
- Scipy

Math & Statistics

- SymPy
- **Statsmodels**

Machine Learning

- **Scikit-Learn**
- Shogun
- PyBrain
- PyLearn2
- PyMC

Plotting & Visualization

- **Bokeh**
- d3py
- ggplot
- **Matplotlib**
- **Plotly**
- prettyplotlib
- **seaborn**

Data Formatting & Storage

- csvkit
- PyTables
- sqlite3

Deep Learning

- **TensorFlow**
- **Keras**
- PyTorch (FaceBook)
- Theano (discontinued by Prof. Y Bengio)

The NumPy Package

http://www.numpy.org/

NumPy (Numerical Python) is the fundamental **package** for **scientific computing with Python**. It contains among other things:

- a powerful N-dimensional array object (multi-dimensional arrays and matrices)
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is licensed under the BSD license, enabling reuse with few restrictions.

The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In **2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications**. NumPy is open-source software and has many contributors.

Initial release	As Numeric, 1995; as NumPy, 2006
Stable release	1.14.0 / 6 January 2018
Written in	Python / C

Numpy Arrays

Numpy arrays are alternatives to Python **Lists**. Key advantages of Numpy arrays are: fast, easy to work with, and give users the opportunity to **perform calculations across entire arrays**.

```
# Create 2 new lists height and weight
height = [1.87,  1.87, 1.82, 1.91, 1.90, 1.85]
weight = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

# Import the numpy package as np
import numpy as np

# Create 2 numpy arrays from height and weight
np_height = np.array(height)
np_weight = np.array(weight)

# print out the type of np_height
print(type(np_height))
```

Element-wise calculations

```
# Calculate bmi (Body Mass Index)
bmi = np_weight / np_height ** 2

# Print the result
print(bmi)
```

We can perform **element-wise calculations** on height and weight. For example, you could take all 6 of the height and weight observations above, and **calculate the BMI for each observation with a single equation**. These operations are **very fast and computationally efficient.** They are particularly helpful when you have 1000s of observations in your data.

Numpay Arrays --- cnt'd

Subsetting

Another great feature of Numpy arrays is the **ability to subset**.

E.g., to know which observations in our BMI array are above 23, we could quickly subset it to find out.

```
# For a boolean response
bmi > 23

# Print only those observations above 23
bmi[bmi > 23]
```

Exercise

1) Convert the list of weights from a list to a Numpy array.
2) Convert all of the weights to pounds. (Use the scalar conversion of 2.2 lbs per kilogram)
3) Print the resulting array of weights in pounds.

```
weight_kg = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

import numpy as np

# Create a numpy array np_weight_kg from weight_kg

# Create np_weight_lbs from np_weight_kg

# Print out np_weight_lbs
```

```
weight_kg = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]
import numpy as np
np_weight_kg = np.array(weight_kg)
np_weight_lbs = np_weight_kg * 2.2
print(np_weight_lbs)
```

The SciPy Library

https://www.scipy.org/scipylib/index.html

The SciPy library is one of the core packages that make up the **SciPy stack.** SciPy is an open source Python library used for **scientific** computing and **technical computing**. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT (Fast Fourier Transformations), signal and image processing, ODE (Ordinary Differential Equations) solvers and other tasks common in science and engineering.

SciPy **builds on the NumPy array object** and is part of the SciPy stack which includes tools like Matplotlib, pandas and SymPy (see below), expanding set of scientific computing libraries.

Original author(s)	Travis Oliphant, Pearu Peterson, Eric Jones
Initial release	Around 2001
Stable release	1.7.1 / 01 August 2021
Written in	Python, Fortran, C, C++

The **SciPy stack/ecosystem** with some of its core packages

SciPy Structure (from SciPy Developer's Guide)

All SciPy modules should follow the following conventions. In the following, a *SciPy module* is defined as a Python package, say `yyy`, that is located in the scipy/ directory.

- Ideally, each SciPy module should be as self-contained as possible. That is, it should have minimal dependencies on other packages or modules. Even dependencies on other SciPy modules should be kept to a minimum. A dependency on NumPy is of course assumed.
- Directory `yyy/` contains:
  - A file `setup.py` that defines `configuration(parent_package='',top_path=None)` function for numpy.distutils.
  - A directory `tests/` that contains files `test_<name>.py` corresponding to modules `yyy/<name>{.py,.so,/}`.
- Private modules should be prefixed with an underscore `_`, for instance `yyy/_somemodule.py`.
- User-visible functions should have good documentation following the Numpy documentation style, see how to document
- The `__init__.py` of the module should contain the main reference documentation in its docstring. This is connected to the Sphinx documentation under `doc/` via Sphinx's automodule directive.

  The reference documentation should first give a categorized list of the contents of the module using `autosummary::` directives, and after that explain points essential for understanding the use of the module.

  Tutorial-style documentation with extensive examples should be separate, and put under `doc/source/tutorial/`

See the existing Scipy submodules for guidance.

For further details on Numpy distutils, see:

> https://github.com/numpy/numpy/blob/master/doc/DISTUTILS.rst.txt

The SciPy API

API - importing from Scipy

In Python the distinction between what is the public API of a library and what are private implementation details is not clear. Unlike in other languages like Java, it is possible in Python to access “private” function or objects. Occasionally this may be convenient, but be aware that if you do so your code may break without warning in future releases. Some widely understood rules for what is and isn’t public in Python are:

- Methods / functions / classes and module attributes whose names begin with a leading underscore are private.
- If a class name begins with a leading underscore none of its members are public, whether or not they begin with a leading underscore.
- If a module name in a package begins with a leading underscore none of its members are public, whether or not they begin with a leading underscore.
- If a module or package defines __all__ that authoritatively defines the public interface.
- If a module or package doesn’t define __all__ then all names that don’t start with a leading

Note

Reading the above guidelines one could draw the conclusion that every private module or object starts with an underscore. This is not the case; **the presence of underscores do mark something as private, but the absence of underscores do not mark something as public**.

In Scipy there are modules whose names don’t start with an underscore, but that should be considered private. To clarify which modules these are we define what the public API is for Scipy, and give some recommendations for how to import modules/functions/objects from Scipy.

Guidelines for importing functions from SciPy

The scipy namespace itself only contains functions imported from numpy. These functions still exist for backwards compatibility, but should be imported from numpy directly.

Everything in the namespaces of scipy submodules is public. In general, it is recommended to import functions from **submodule** namespaces. For example, the function **curve_fit** (defined in scipy/optimize/minpack.py) should be imported like this:

```
from scipy import optimize
result = optimize.curve_fit(...)
```

This **form of importing submodules is preferred** for all submodules except scipy.io (because io is also the name of a module in the Python s

```
from scipy import interpolate
from scipy import integrate
import scipy.io as spio
```

In some cases, the public API is one level deeper.

For example the **scipy.sparse.linalg module is public, and the functions it contains are not available in the scipy.sparse namespace**. Sometimes it may result in more easily understandable code if functions are imported from one level deeper.

For example, below it is immediately clear that lomax is a distribution if the second form is chosen:

```
# first form
from scipy import stats
stats.lomax(...)

# second form
from scipy.stats import distributions
distributions.lomax(...)
```

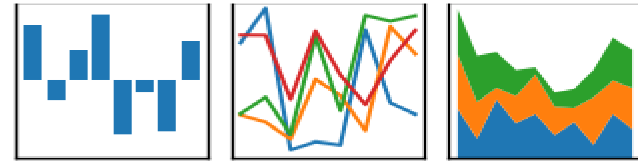
In that case the second form can be chosen, if it is documented in the next section that the submodule in question is public.

Pandas – The Python Data Analysis Library

https://pandas.pydata.org/

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$



Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Pandas is a NumFOCUS sponsored project. It offers data structures and operations for manipulating **numerical tables** and **time series**. Its name is derived from the term "**panel data**", an econometrics term for data sets that include both time-series and cross-sectional data.

Original author(s)	Wes McKinney (*)
Stable release	1.3.3 / 12 September 2021
Repository	https://github.com/pandas-dev/pandas
Written in	Python

(*) Developer Wes McKinney started working on pandas in 2008 while at AQR Capital Management out of the need for a **high performance, flexible tool to perform quantitative analysis on financial data**. Before leaving AQR he was able to convince management to allow him to open source the library.

A very useful video by Wes McKinney on Pandas:
10-Minute Tour of Pandas:
https://vimeo.com/59324550

Pandas Dataframe Object

Pandas is an excellent *data-processing* library

```
import pandas
df = pandas.DataFrame({
        'A': [-137, 22, -3, 4, 5],
        'B': [10, 11, 121, 13, 14],
        'C': [3, 6, 91, 12, 15],
    })
```

df is what Pandas calls a **dataframe**

```
>>> print(df)
     A    B   C
0 -137   10   3
1   22   11   6
2   -3  121  91
3    4   13  12
4    5   14  15
```

Filtering

You can filter out rows in dataframe,
to get another, smaller dataframe

```
>>> positive_a = df[df.A > 0]
>>> print(positive_a)
    A   B   C
1  22  11   6
3   4  13  12
4   5  14  15
```

This is strange

Look again at the code:

```
positive_a = df[df.A > 0]
```

The expression **df.A > 0** ought to be True or False, correct?
So there would be no way to filter rows dynamically at run time.

How does this even work????.

Pandas Dataframe Filtering

Is this cheating??

It turns out it is NOT Boolean at all:

```
>>> comparison = (df.A > 0)
>>> type(comparison)
<class 'pandas.core.series.Series'>
>>> print(comparison)
0     False
1      True
2     False
3      True
4      True
Name: A, dtype: bool
```

Comparison Object

The expression **df.A > 0** is translated to **df.A.__gt__(0)**
Rather than re-inventing Pandas, let's create a similar , but simplified library. If df.A represents a data column, let's have a Column type whose __gt__ method returns a Comparison object.

See next slide for the **__gt__** method

```
import operator
class Column:
    def __init__(self, name):
        self.name = name
    def __gt__(self, value):
        return Comparison(self.name, value, operator.gt)
```

More Complex Expressions

This is how you might implement a Pandas-like interface.
To evaluate expressions like **df[df.C + 2 < df.B]**, you need to do more work – but it can all be done via these 'magic' methods.

PACE UNIVERSITY

Python’s Special Methods

There exist “special methods” — certain **“magic” methods** — that Python invokes when you use certain syntax. Using special methods, your classes can act like sets, like dictionaries, like functions, like iterators, or even like numbers. The most common special method: the __init__() method.

Classes that can be Compared

If you create your own class it makes sense to compare your objects to other objects, you can use the following special methods to **implement comparisons**.

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> y = Fraction(1, 2)
>>> z = Fraction(1, 4)
>>>
>>> x.__gt__(y)
False
>>>
>>> x.__gt__(z)
True
>>>
```

Note
The **fractions** module
(lower case ‘f’)
The **Fraction** instance
(upper case ‘F’)

You Want...	So You Write...	And Python Calls...
equality	`x == y`	`x.__eq__(y)`
inequality	`x != y`	`x.__ne__(y)`
less than	`x < y`	`x.__lt__(y)`
less than or equal to	`x <= y`	`x.__le__(y)`
greater than	`x > y`	`x.__gt__(y)`
greater than or equal to	`x >= y`	`x.__ge__(y)`
truth value in a boolean context	`if x:`	`x.__bool__()`

PACE UNIVERSITY

Pandas Basics

Pandas DataFrames

Pandas is a high-level **data manipulation** tool. It is built on top of the Numpy package and its key data structure is called **DataFrame**. DataFrames allow you to **store** and **manipulate tabular** data in **rows** of observations and **columns** of variables.

There are many ways to create a DataFrame. One way is to use a dictionary**.**

```
dict = {"country": ["Brazil", "Russia", "India", "China", "South Africa"],
       "capital": ["Brasilia", "Moscow", "New Delhi", "Beijing", "Pretoria"],
       "area": [8.516, 17.10, 3.286, 9.597, 1.221],
       "population": [200.4, 143.5, 1252, 1357, 52.98] }
```

```
import pandas as pd
brics = pd.DataFrame(dict)
print(brics)
```

As you see the new brics DataFrame, Pandas has assigned a key for each country, numerical values 0 through 4.

```
>>> import pandas as pd
>>> dict = {"country": ["Brazil", "Russia", "India", "China", "South Africa"],
...         "capital": ["Brasilia", "Moscow", "New Delhi", "Beijing", "Pretoria"]
,
...         "area": [8.516, 17.10, 3.286, 9.597, 1.221],
...         "population": [200.4, 143.5, 1252, 1357, 52.98] }
>>> brics = pd.DataFrame(dict)
>>> print(brics)
     area    capital       country  population
0   8.516   Brasilia        Brazil      200.40
1  17.100     Moscow        Russia      143.50
2   3.286  New Delhi         India     1252.00
3   9.597    Beijing         China     1357.00
4   1.221   Pretoria  South Africa       52.98
>>>
```

Pandas Basics … cnt'd

In the previous example you could have different index values, say, the two letter country code:

```
# Set the index for brics
brics.index = ["BR", "RU", "IN", "CH", "SA"]

# Print out brics with new index values
print(brics)
```

	area	capital	country	population
BR	8.516	Brasilia	Brazil	200.40
RU	17.100	Moscow	Russia	143.50
IN	3.286	New Dehli	India	1252.00
CH	9.597	Beijing	China	1357.00
SA	1.221	Pretoria	South Africa	52.98

Another way to **create a DataFrame** is by **importing a csv file** using Pandas.

```
# Import pandas as pd
import pandas as pd

# Import the cars.csv data: cars
cars = pd.read_csv('cars.csv')

# Print out cars
print(cars)
```

	Unnamed: 0	cars_per_cap	country	drives_right
0	US	809	United States	True
1	AUS	731	Australia	False
2	JAP	588	Japan	False
3	IN	18	India	False
4	RU	200	Russia	True
5	MOR	70	Morocco	True
6	EG	45	Egypt	True

Pandas Basics … cnt'd

Indexing Pandas DataFrames

There are many ways to index a DataFrame. The easiest way is to use **square bracket notation.**

In the example below, you can use **square brackets to select one column** of the cars DataFrame. You can either use a single bracket or a double bracket. The **single bracket** will output a **Pandas Series**, while a **double bracket** will output a **Pandas DataFrame**.

```
# Import pandas and cars.csv
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

# Print out country column as Pandas Series (single bracket
print(cars['cars_per_cap'])

# Print out country column as Pandas DataFrame (double…)
print(cars[['cars_per_cap']])

# Print out DataFrame with country and drives_right columns
print(cars[['cars_per_cap', 'country']])
```

```
output:
   US     809
   AUS    731
   JAP    588
   IN      18
   RU     200
   MOR     70
   EG      45
   Name: cars_per_cap, dtype: int64
        cars_per_cap
   US            809
   AUS           731
   JAP           588
   IN             18
   RU            200
   MOR            70
   EG             45
        cars_per_cap        country
   US            809  United States
   AUS           731      Australia
   JAP           588          Japan
   IN             18          India
   RU            200         Russia
   MOR            70        Morocco
   EG             45          Egypt
```

PACE UNIVERSITY

Pandas Basics … cnt'd

Square brackets can also be used to access **observations** (**rows**) from a **DataFrame**

```
# Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

# Print out first 4 observations
print(cars[0:4])

# Print out fifth, sixth, and seventh observation
print(cars[4:6])
```

	cars_per_cap	country	drives_right
US	809	United States	True
AUS	731	Australia	False
JAP	588	Japan	False
IN	18	India	False

	cars_per_cap	country	drives_right
RU	200	Russia	True
MOR	70	Morocco	True

Pandas Basics … cnt'd

You can also use **loc** and **iloc** to perform just about any data selection operation.

loc is **label-based**, which means that you have to specify rows and columns based on their row and column labels.

iloc is **integer index based**, so you have to specify rows and columns by their integer index like you did in the previous exercise.

```
# Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

#Print out observation for Japan
Print(cars.iloc[2])

#Print out observation for Australia and Egy
print(cars.loc[['AUS', 'EG']])
```

```
cars_per_cap      588
country         Japan
drives_right    False
Name: JAP, dtype: object
     cars_per_cap    country drives_right
AUS           731  Australia        False
EG             45      Egypt         True
```

Python's Pandas library Example

- Install the pandas-datareader library
- https://github.com/pydata/pandas-datareader
- $pip install pandas-datareader

Remote Data Access by using Pandas Datareader

http://pandas-datareader.readthedocs.io/en/latest/remote_data.html

Functions from `pandas_datareader.data` and `pandas_datareader.wb` extract data from various Internet sources into a pandas DataFrame. Currently the following sources are supported:

- Yahoo! Finance
- Google Finance
- Enigma
- Quandl
- St.Louis FED (FRED)
- Kenneth French's data library
- World Bank
- OECD
- Eurostat
- Thrift Savings Plan
- Nasdaq Trader symbol definitions

It should be noted, that various sources support different kinds of data, so not all sources implement the same methods and the data elements returned might also differ.

Python's Pandas library Example (Jupyter Notebook)

Run the Jupyter Notebook with the Pandas' **pandas-datareader** module.

```
from pandas_datareader import data
import matplotlib.pyplot as plt
import pandas as pd
```

```
# Define the instruments to download. We would like to see Apple, Microsoft and the
S&P500 index.
tickers = ['AAPL', 'MSFT', '^GSPC']

# Define which online source one should use
data_source = 'yahoo'

# We would like all available data from 01/01/2000 until 12/31/2016.
start_date = '2000-01-01'
end_date = '2016-12-31'

# User pandas_reader.data.DataReader to load the desired data. As simple as that.
panel_data = data.DataReader(tickers, data_source, start_date, end_date)
```

```
>>> all_weekdays
DatetimeIndex(['2000-01-03', '2000-01-04', '2000-01-05', '2000-01-06',
               '2000-01-07', '2000-01-10', '2000-01-11', '2000-01-12',
               '2000-01-13', '2000-01-14',
               ...
               '2016-12-19', '2016-12-20', '2016-12-21', '2016-12-22',
               '2016-12-23', '2016-12-26', '2016-12-27', '2016-12-28',
               '2016-12-29', '2016-12-30'],
              dtype='datetime64[ns]', length=4435, freq='B')
```

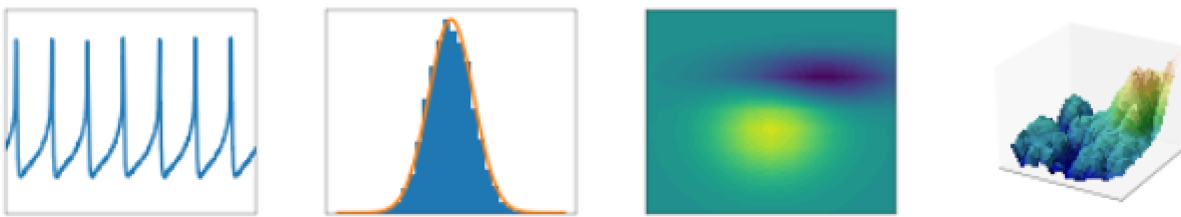
Python's Matplotlib library

- Install the matplotlib library
- http://matplotlib.org
- $pip install matplotlib

matplotlib

Introduction

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.

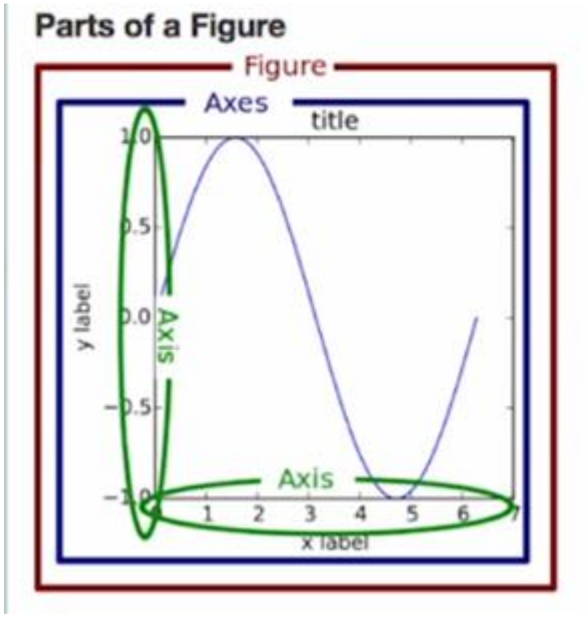


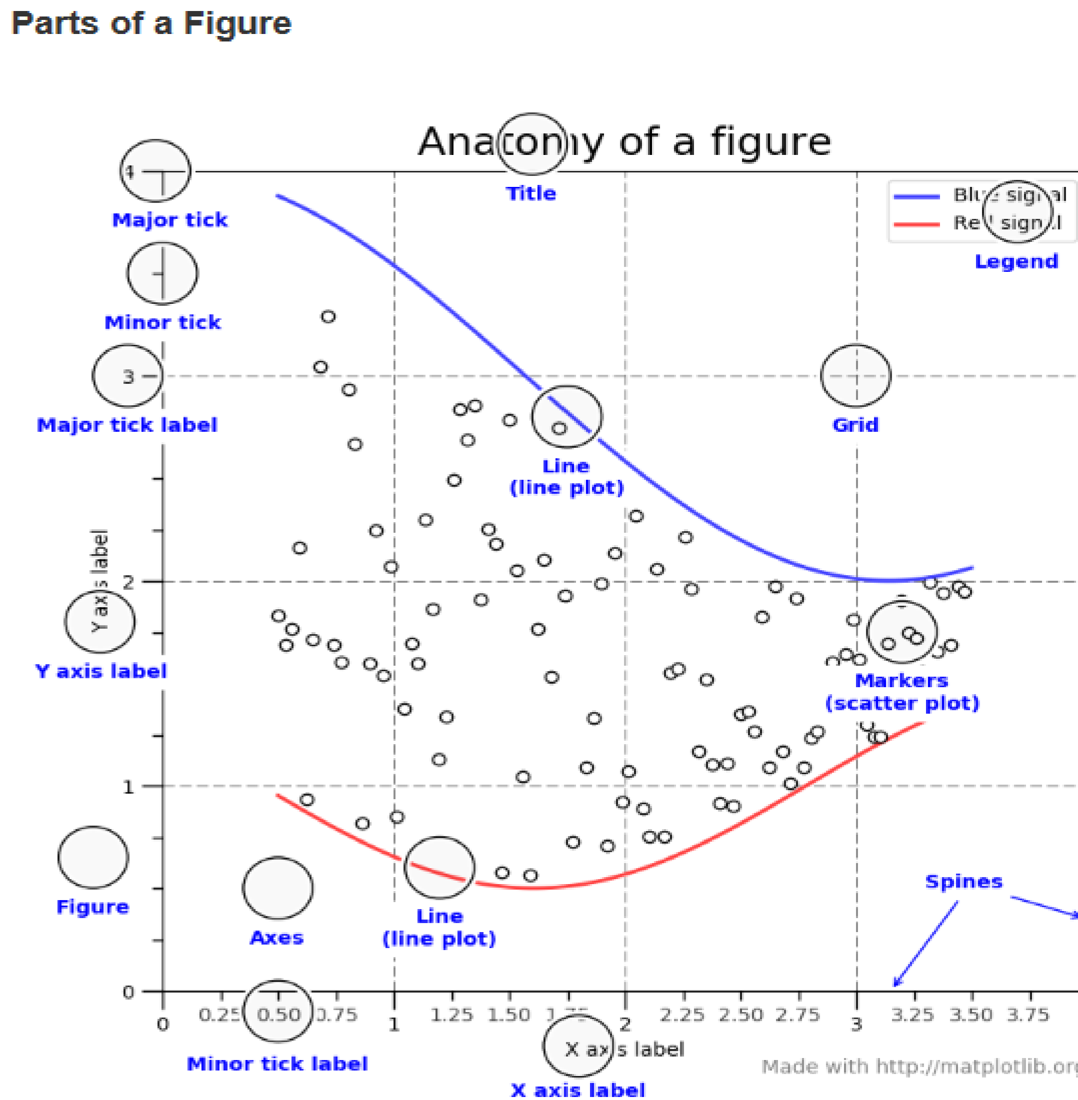
Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For a sampling, see the screenshots, thumbnail gallery, and examples directory

For simple plotting the `pyplot` module provides a MATLAB-like interface, particularly when combined with `IPython`. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Python’s Matplotlib library Example (Jupyter Notebook)

Run the Jupyter Notebook with the Pandas’ **matplotlib** module.





http://matplotlib.org/faq/usage_faq.html

PACE UNIVERSITY

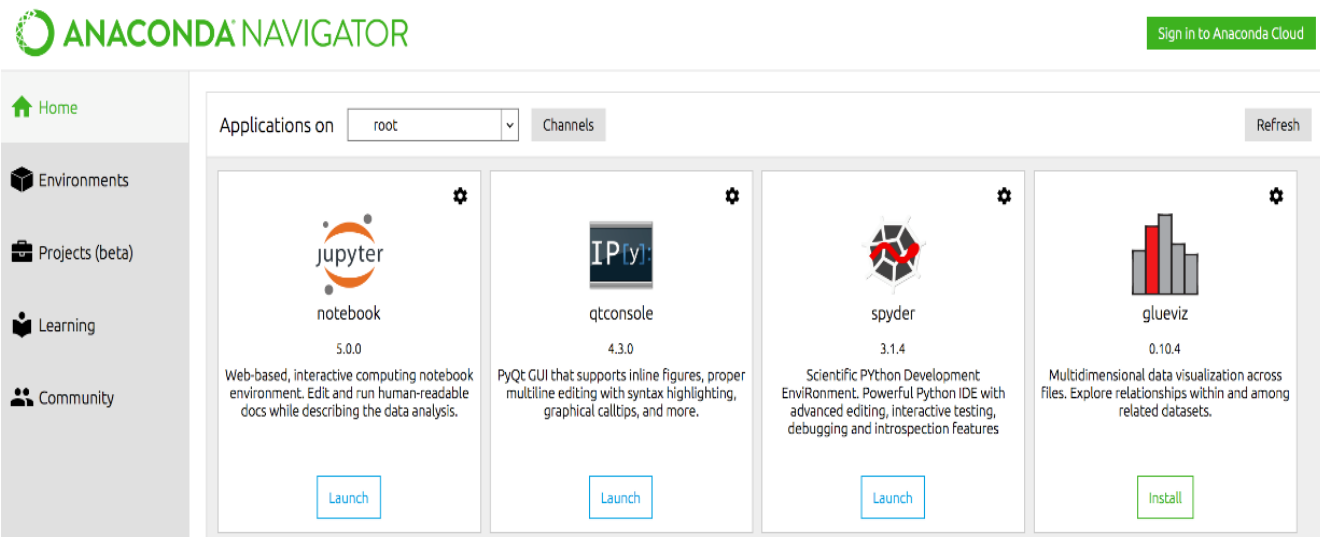
Python Advanced Topics

- Generators
- List Comprehensions
- Multiple Function Arguments
- Regular Expressions
- Exception Handling
- Sets
- Serialization (see pickle! SerDe: Serialization/Deserialization)
- Partial functions
- Code Introspection
- Closures
- Decorators

Anaconda (Python) Data Science Platform

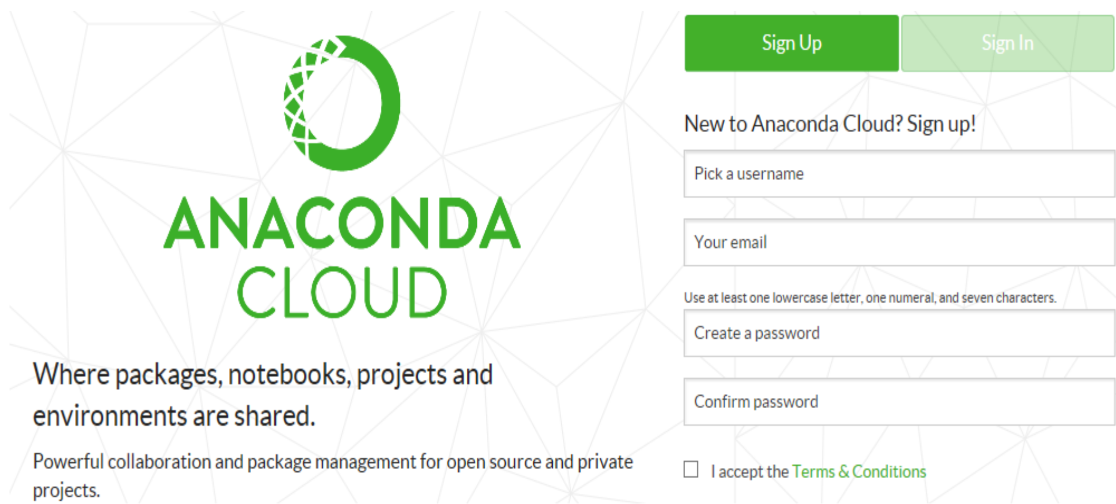
Anaconda Navigator

https://docs.anaconda.com/anaconda/navigator/



Anaconda Cloud

https://anaconda.org/



PACE UNIVERSITY

Questions?