Introduction

Python is a high-level, interpreted, object-oriented programming language with dynamic semantics as an added advantage. The combination of high-level built-in data structures with dynami typing and dynamic binding makes it very attractive for Rapid Application Development, as well as for scripting or glue language to connect to the exisiting components. Python symphasizes readability and reduces the cost of the program maintenance by its outstanding feature simple and easy to learn syntax.Program modularity and code reuse can be acheieved by Python modules and packages.The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Python is a great general-purpose programming language on its own. But it has become even more popular environment for scientific computing with the help of a few popular libraries (numpy, scipy, matplotlib).

Python Version

Python 2

Published in late 2000, Python 2 signalled a more transparent and inclusive language development process than earlier versions of Python with the implementation of PEP (Python Enhancement Proposal), a technical specification that either provides information to Python community members or describes a new feature of the language.

Additionally, Python 2 included many more programmatic features including a cycle-detecting garbage collector to automate memory management, increased Unicode support to standardize characters, and list comprehensions to create a list based on existing lists. As Python 2 continued to develop, more features were added, including unifying Python’s types and classes into one hierarchy in Python version 2.2.

Python 3

Python 3 is regarded as the future of Python and is the version of the language that is currently in development. A major overhaul, Python 3 was released in late 2008 to address and amend intrinsic design flaws of previous versions of the language. The focus of Python 3 development was to clean up the codebase and remove redundancy, making it clear that there was only one way to perform a given task.

Major modifications to Python 3.0 included changing the print statement into a built-in function, improve the way integers are divided, and providing more Unicode support.

At first, Python 3 was slowly adopted due to the language not being backwards compatible with Python 2, requiring people to make a decision as to which version of the language to use. Additionally, many package libraries were only available for Python 2, but as the development team behind Python 3 has reiterated that there is an end of life for Python 2 support, more libraries have

been ported to Python 3. The increased adoption of Python 3 can be shown by the number of Python packages that now provide Python 3 support, which at the time of writing includes 339 of the 360 most popular Python packages.

Python 2.7

Following the 2008 release of Python 3.0, Python 2.7 was published on July 3, 2010 and planned as the last of the 2.x releases. The intention behind Python 2.7 was to make it easier for Python 2.x users to port features over to Python 3 by providing some measure of compatibility between the two. This compatibility support included enhanced modules for version 2.7 like unittest to support test automation, argparse for parsing command-line options, and more convenient classes in collections.

Because of Python 2.7's unique position as a version in between the earlier iterations of Python 2 and Python 3.0, it has persisted as a very popular choice for programmers due to its compatibility with many robust libraries. When we talk about Python 2 today, we are typically referring to the Python 2.7 release as that is the most frequently used version.

Python 2.7, however, is considered to be a legacy language and its continued development, which today mostly consists of bug fixes, will cease completely in 2020.

Key Differences

While Python 2.7 and Python 3 share many similar capabilities, they should not be thought of as entirely interchangeable. Though you can write good code and useful programs in either version, it is worth understanding that there will be some considerable differences in code syntax and handling.

Anaconda Python Distribution

Anaconda is an open-source package manager, environment manager, and distribution of the Python and R programming languages. It is commonly used for large-scale data processing, scientific computing, and predictive analytics, serving data scientists, developers, business analysts, and those working in DevOps.

Anaconda offers a collection of over 720 open-source packages, and is available in both free and paid versions. The Anaconda distribution ships with the conda command-line utility. You can learn more about Anaconda and conda by reading the Anaconda Documentation pages.

Why Anaconda?

- User level install of the version of python you want
- Able to install/update packages completely independent of system libraries or admin privileges
- conda tool installs binary packages, rather than requiring compile resources like pip - again, handy if you have limited privileges for installing necessary libraries.
- More or less eliminates the headaches of trying to figure out which version/release of package X is compatible with which version/release of package Y, both of which are

required for the install of package Z
- Comes either in full-meal-deal version, with numpy, scipy, PyQt, spyder IDE, etc. or in minimal / alacarte version (miniconda) where you can install what you want, when you need it
- No risk of messing up required system libraries

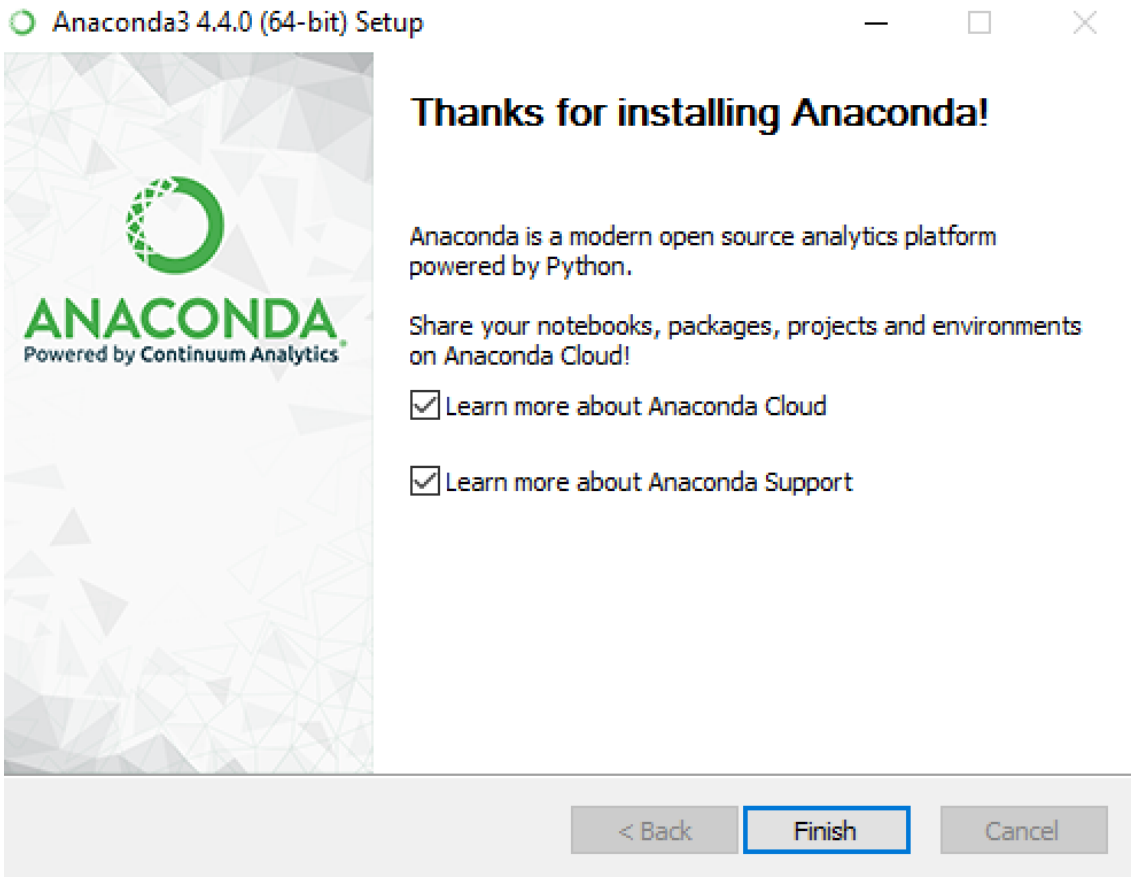
Installing on Windows

1. Download the Anaconda installer (https://www.continuum.io/downloads).
2. Optional: Verify data integrity with MD5 or SHA-256. More info on hashes
3. Double click the installer to launch.

   NOTE: If you encounter any issues during installation, temporarily disable your anti-virus software during install, then re-enable it after the installation concludes. If you have installed for all users, uninstall Anaconda and re-install it for your user only and try again.
4. Click Next.
5. Read the licensing terms and click I Agree.
6. Select an install for “Just Me” unless you’re installing for all users (which requires Windows Administrator privileges).
7. Select a destination folder to install Anaconda and click Next.

   NOTE: Install Anaconda to a directory path that does not contain spaces or unicode characters.

   NOTE: Do not install as Administrator unless admin privileges are required.
8. Choose whether to add Anaconda to your PATH environment variable. We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Command Prompt from the Start Menu.
9. Choose whether to register Anaconda as your default Python 3.6. Unless you plan on installing and running multiple versions of Anaconda, or multiple versions of Python, you should accept the default and leave this box checked.
10. Click Install. You can click Show Details if you want to see all the packages Anaconda is installing.
11. Click Next.
12. After a successful installation you will see the “Thanks for installing Anaconda” image:

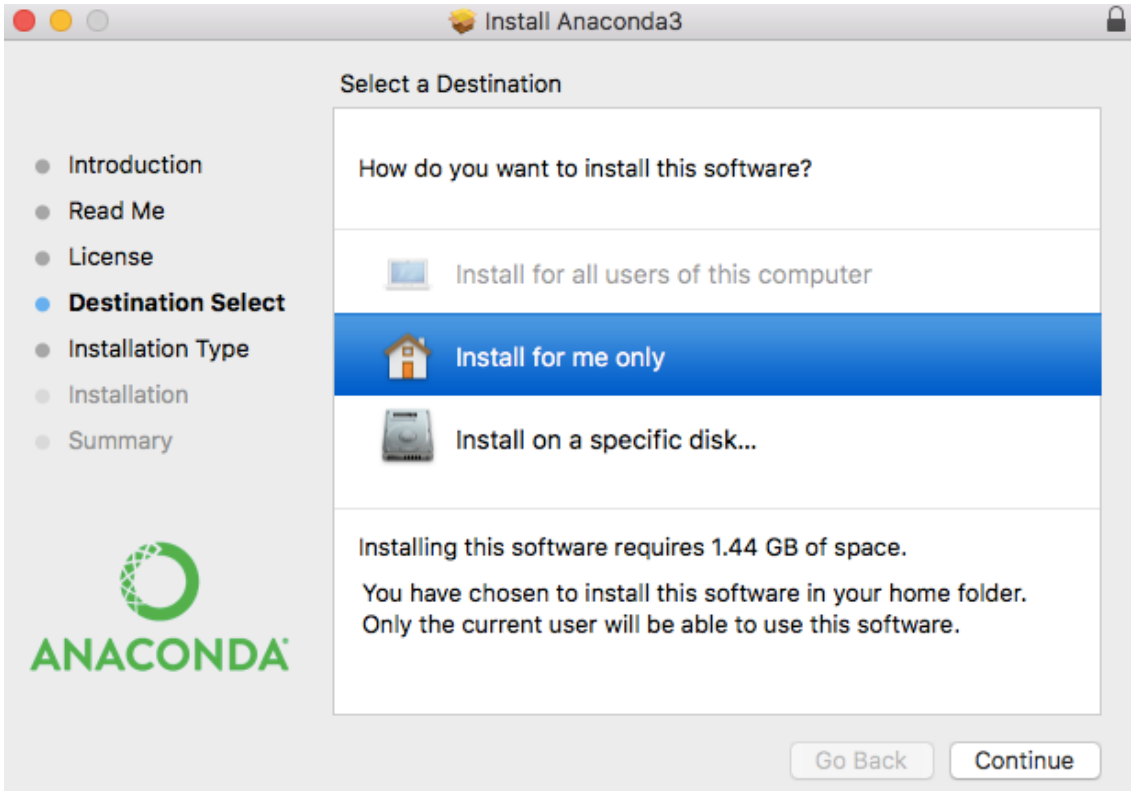


13. You can leave the boxes checked “Learn more about Anaconda Cloud” and “Learn more about Anaconda Support” if you wish to read more about this cloud package management service and Anaconda support. Click Finish.
14. After your install is complete, verify it by opening Anaconda Navigator, a program that is included with Anaconda. From your Windows Start menu, select the shortcut Anaconda Navigator. If Navigator opens, you have successfully installed Anaconda.

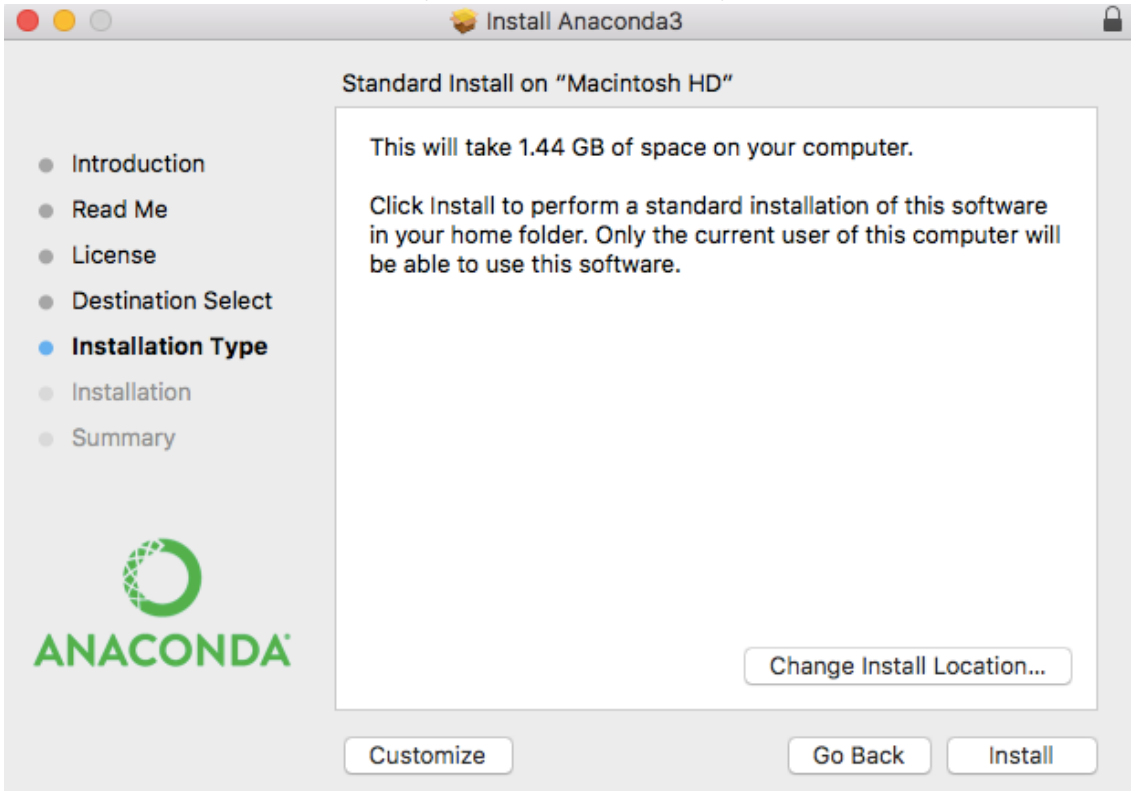
Installing on macOS

1. Download the graphical macOS installer (https://www.continuum.io/downloads#macos) for your version of Python.
2. OPTIONAL: Verify data integrity with MD5 or SHA-256. For more information on hashes, see What about cryptographic hash verification?.
3. Double-click the .pkg file.
4. Answer the prompts on the Introduction, Read Me and License screens.
5. On the Destination Select screen, select Install for me only.

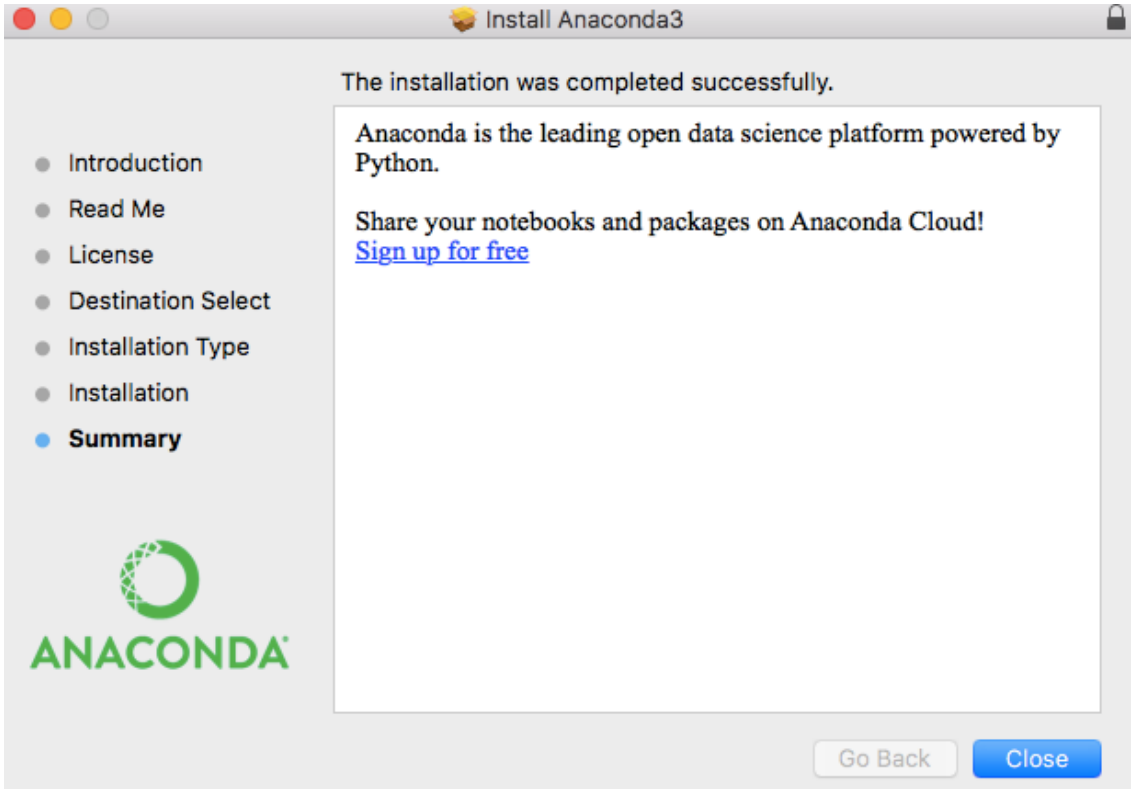
   NOTE: If you get the error message “You cannot install Anaconda in this location,” reselect Install for me only.



6. On the Installation Type screen, you may choose to install in another location. The standard install puts Anaconda in your home user directory:



7. Click the Install button.
8. A successful installation displays the following screen:



Installing on Linux

1. In your browser, download the Anaconda installer for Linux.
2. Optional: Verify data integrity with MD5 or SHA-256. (For more information on hashes, see cryptographic hash validation.)

   Run the following:

   md5sum /path/filename OR:

   sha256sum /path/filename
3. Verify results against the proper hash page to make sure the hashes match.
4. Enter the following to install Anaconda for Python 3.6:

   bash ~/Downloads/Anaconda3-4.4.0-Linux-x86_64.sh OR

   Enter the following to install Anaconda for Python 2.7:

   bash ~/Downloads/Anaconda2-4.4.0-Linux-x86_64.sh NOTE: You should include the bash command regardless of whether you are using Bash shell.

   NOTE: Replace actual download path and filename as necessary.

   NOTE: Install Anaconda as a user unless root privileges are required.
5. The installer prompts “In order to continue the installation process, please review the license agreement.” Click Enter to view license terms.
6. Scroll to the bottom of the license terms and enter yes to agree to them.
7. The installer prompts you to click Enter to accept the default install location, press CTRL-C to cancel the installation, or specify an alternate installation directory. If you accept the default install location, the installer displays **'PREFIX=/home/"user"/anaconda"2 or 3"'**

and continues the installation. It may take a few minutes to complete.

The installer prompts “Do you wish the installer to prepend the Anaconda"2 or 3" install location to PATH in your /home/"user"/.bashrc ?” We recommend yes.

NOTE: If you enter “no”, you will need to manually specify the path to Anaconda when using Anaconda. To manually add the prepended path, edit file .bashrc to add ~/anaconda"2 or 3"/bin to your path manually using:

8. **export PATH="/home/"user"/anaconda"2 or 3"/bin:$PATH"** Replace /home/"user"/anaconda"2 or 3" with the actual path.
9. The installer finishes and displays “Thank you for installing Anaconda"2 or 3"!”
10. Close and open your terminal window for the installation to take effect, or you can enter the command source ~/.bashrc.

    Enter conda list. If the installation was successful, the terminal window should display a list of installed Anaconda packages.

    NOTE: Power8 users: Navigate to your Anaconda directory and run this command:

    conda install libgfortran

What are Jupyter notebook?

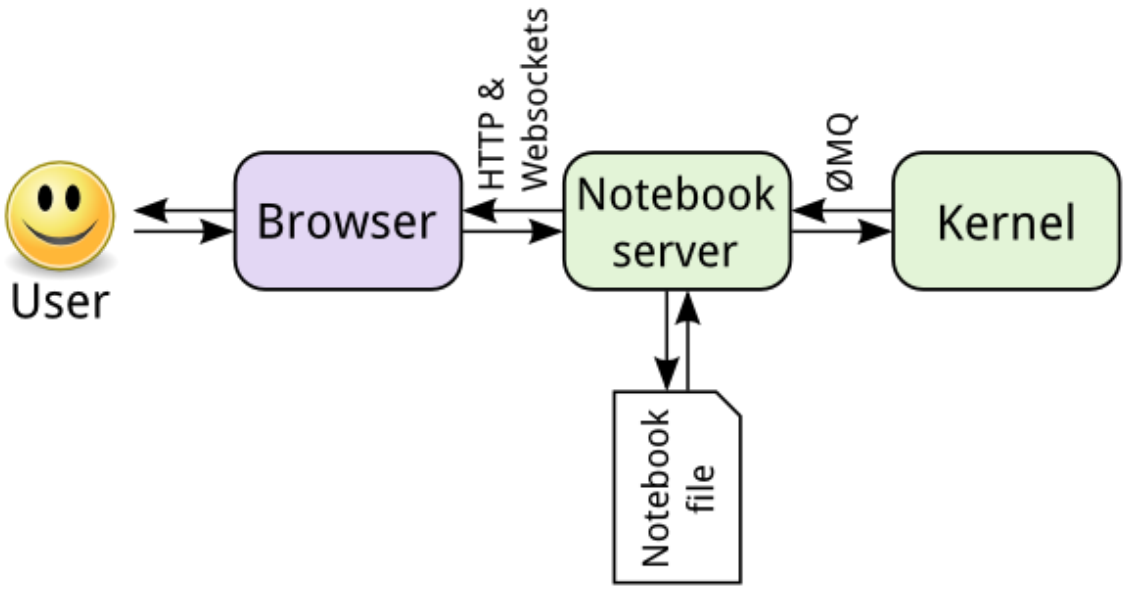
The notebook is a web application that allows you to combine explanatory text, math equations, code, and visualizations all in one easily sharable document.

Notebooks have quickly become an essential tool when working with data. You'll find them being used for data cleaning and exploration, visualization, machine learning, and big data analysis. Typically you'd be doing this work in a terminal, either the normal Python shell or with IPython. Your visualizations would be in separate windows, any documentation would be in separate documents, along with various scripts for functions and classes. However, with notebooks, all of these are in one place and easily read together.

Notebooks are also rendered automatically on GitHub. It’s a great feature that lets you easily share your work. There is also http://nbviewer.jupyter.org/ (http://nbviewer.jupyter.org/) that renders the notebooks from your GitHub repo or from notebooks stored elsewhere.

How notebook works?

Jupyter notebooks grew out of the IPython project started by Fernando Perez. IPython is an interactive shell, similar to the normal Python shell but with great features like syntax highlighting and code completion. Originally, notebooks worked by sending messages from the web app (the notebook you see in the browser) to an IPython kernel (an IPython application running in the background). The kernel executed the code, then sent it back to the notebook. The current architecture is similar, drawn out below.



The central point is the notebook server. You connect to the server through your browser and the notebook is rendered as a web app. Code you write in the web app is sent through the server to the kernel. The kernel runs the code and sends it back to the server, then any output is rendered back in the browser. When you save the notebook, it is written to the server as a JSON file with a .ipynb file extension.

The great part of this architecture is that the kernel doesn't need to run Python. Since the notebook and the kernel are separate, code in any language can be sent between them. For example, two of the earlier non-Python kernels were for the R and Julia languages. With an R kernel, code written in R will be sent to the R kernel where it is executed, exactly the same as Python code running on a Python kernel. IPython notebooks were renamed because notebooks became language agnostic. The new name Jupyter comes from the combination of Julia, Python, and R. If you're interested, here's a list of available kernels.

Another benefit is that the server can be run anywhere and accessed via the internet. Typically you'll be running the server on your own machine where all your data and notebook files are stored. But, you could also set up a server on a remote machine or cloud instance like Amazon's EC2. Then, you can access the notebooks in your browser from anywhere in the world.

Installing Jupyter Notebook

By far the easiest way to install Jupyter is with Anaconda. Jupyter notebooks automatically come with the distribution. You'll be able to use notebooks from the default environment.

To install Jupyter notebooks in a conda environment, use **conda install jupyter notebook**.

Jupyter notebooks are also available through pip with **pip install jupyter notebook**.

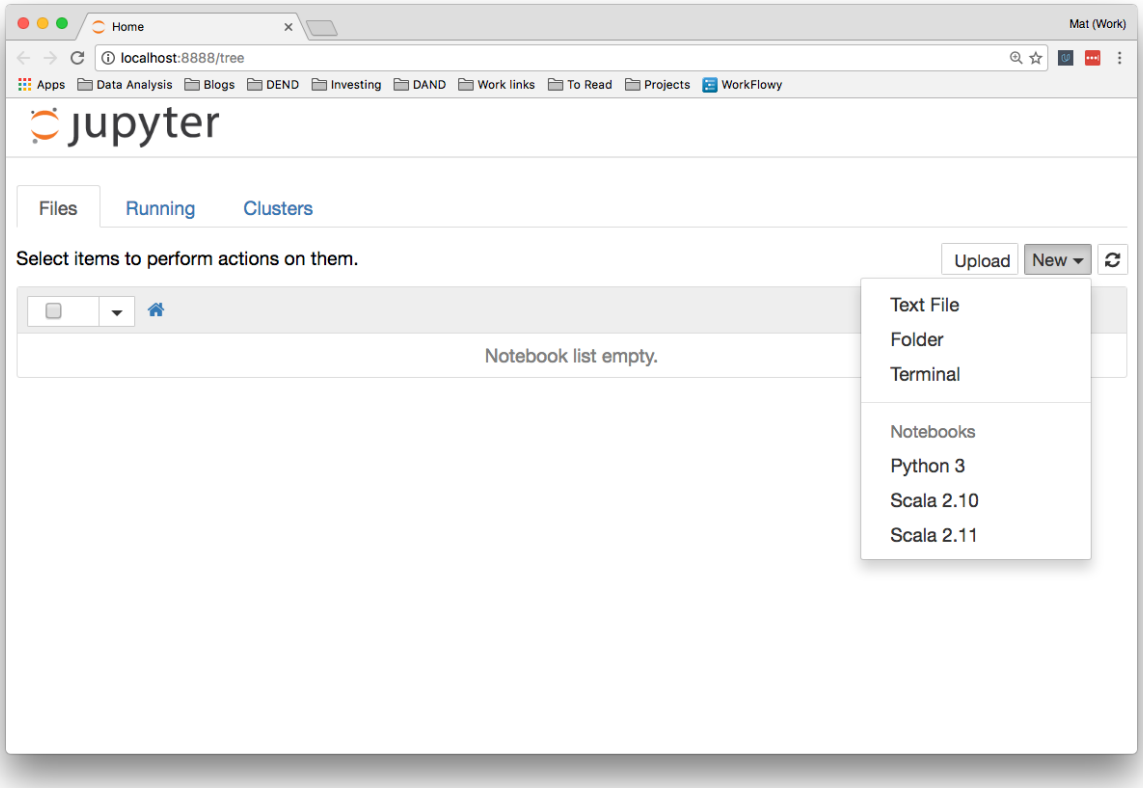
Launching the notebook server

To start a notebook server, enter jupyter notebook in your terminal or console. This will start the server in the directory you ran the command in. That means any notebook files will be saved in that directory. Typically you'd want to start the server in the directory where your notebooks live. However, you can navigate through your file system to where the notebooks are.

When you run the command (try it yourself!), the server home should open in your browser. By default, the notebook server runs at http://localhost:8888 (http://localhost:8888). If you aren't familiar with this, localhost means your computer and 8888 is the port the server is communicating on. As long as the server is still running, you can always come back to it by going to http://localhost:8888 (http://localhost:8888) in your browser.

If you start another server, it'll try to use port 8888, but since it is occupied, the new server will run on port 8889. Then, you'd connect to it at http://localhost:8889 (http://localhost:8889). Every additional notebook server will increment the port number like this.

If you tried starting your own server, it should look something like this:



Basics of Python

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

classic quicksort algorithm in Python:

Fundamental types

Integers

Integer literals are created by any number without a decimal or complex component.

In [19]:
```
# integers
x = 1
type(x)
```

Out[19]:
```
int
```

Floats

Float literals can be created by adding a decimal component to a number.

In [2]:
```
# float
x = 1.0
type(x)
```

Out[2]:
```
float
```

Boolean

Boolean can be defined by typing True/False without quotes

In [3]:
```
# boolean
b1 = True
b2 = False

type(b1)
```

Out[3]:
```
bool
```

Strings

String literals can be defined with any of single quotes ('), double quotes (") or triple quotes (''' or """). All give the same result with two important differences.

If you quote with single quotes, you do not have to escape double quotes and vice-versa. If you quote with triple quotes, your string can span multiple lines.

In [4]:
```
# string
name = 'your name'

type(name)
```

Out[4]:
```
str
```

Complex

Complex literals can be created by using the notation x + yj where x is the real component and y is the imaginary component.

In [5]:
```
# complex numbers: note the use of `j` to specify the imaginary part
x = 1.0 - 1.0j
type(x)
```

Out[5]:
```
complex
```

In [23]:
```
print(x)
```

```
(1-1j)
```

In [24]:
```
print(x.real, x.imag)
```

```
(1.0, -1.0)
```

Variables

Definining

A variable in Python is defined through assignment. There is no concept of declaring a variable outside of that assignment.

In [6]:
```
ten = 10
ten
```

Out[6]:
```
10
```

Dynamic Typing

In Python, while the value that a variable points to has a type, the variable itself has no strict type in its definition. You can re-use the same variable to point to an object of a different type. It may be helpful to think of variables as "labels" associated with objects.

In [7]:
```
ten = 10
ten
```

Out[7]:
```
10
```

In [8]:
```
ten = 'ten'
ten
```

Out[8]:
```
'ten'
```

Strong Typing

While Python allows you to be very flexible with your types, you must still be aware of what those types are. Certain operations will require certain types as arguments.

In [9]:
```
'Day ' + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-0ff57d40f668> in <module>()
----> 1 'Day ' + 1

TypeError: Can't convert 'int' object to str implicitly
```

This behavior is different from some other loosely-typed languages. If you were to do the same thing in JavaScript, you would get a different result.

In Python, however, it is possible to change the type of an object through builtin functions.

In [10]:
```
'Day ' + str(1)
```

Out[10]:
```
'Day 1'
```

Simple Expressions

Boolean Evaluation

Boolean expressions are created with the keywords and, or, not and is. For example:

In [14]:
```
True and False
```

Out[14]:
```
False
```

In [15]:
```
True or False
```

Out[15]:
```
True
```

In [16]:
```
not True
```

Out[16]:
```
False
```

In [17]:
```
not False
```

Out[17]:
```
True
```

In [18]:
```
True is True
```

Out[18]: True

In [19]:
```
True is False
```

Out[19]: False

In [20]:
```
'a' is 'a'
```

Out[20]: True

Branching (if / elif / else)

Python provides the if statement to allow branching based on conditions. Multiple elif checks can also be performed followed by an optional else clause. The if statement can be used with any evaluation of truthiness.

In [26]:
```
i = 3
if i < 3:
    print('less than 3')
elif i < 5:
    print('less than 5')
else:
    print('5 or more')
```

```
less than 5
```

Block Structure and Whitespace

The code that is executed when a specific condition is met is defined in a "block." In Python, the block structure is signalled by changes in indentation. Each line of code in a certain block level must be indented equally and indented more than the surrounding scope. The standard (defined in PEP-8) is to use 4 spaces for each level of block indentation. Statements preceding blocks generally end with a colon (:).

Because there are no semi-colons or other end-of-line indicators in Python, breaking lines of code requires either a continuation character (\ as the last char) or for the break to occur inside an unfinished structure (such as open parentheses).

Advanced Types: Containers

One of the great advantages of Python as a programming language is the ease with which it allows you to manipulate containers. Containers (or collections) are an integral part of the language and, as you'll see, built in to the core of the language's syntax. As a result, thinking in a Pythonic manner means thinking about containers.

Lists

The first container type that we will look at is the list. A list represents an ordered, mutable collection of objects. You can mix and match any type of object in a list, add to it and remove from it at will.

Creating Empty Lists. To create an empty list, you can use empty square brackets or use the list() function with no arguments.

In [27]:
```
l = []
l
```

Out[27]: []

In [28]:
```
l = list()
l
```

Out[28]: []

Initializing Lists. You can initialize a list with content of any sort using the same square bracket notation. The list() function also takes an iterable as a single argument and returns a shallow copy of that iterable as a new list. A list is one such iterable as we'll see soon, and we'll see others later.

In [29]:
```
l = ['a', 'b', 'c']
l
```

Out[29]: ['a', 'b', 'c']

In [30]:
```
l2 = list(l)
l2
```

Out[30]: ['a', 'b', 'c']

A Python string is also a sequence of characters and can be treated as an iterable over those characters. Combined with the list() function, a new list of the characters can easily be generated.

In [31]:
```
list('abcdef')
```

Out[31]: ['a', 'b', 'c', 'd', 'e', 'f']

Adding. You can append to a list very easily (add to the end) or insert at an arbitrary index.

In [32]:
```
l = []
l.append('b')
l.append('c')
l.insert(0, 'a')
l
```

Out[32]: ['a', 'b', 'c']

Iterating. Iterating over a list is very simple. All iterables in Python allow access to elements using the for ... in statement. In this structure, each element in the iterable is sequentially assigned to the "loop variable" for a single pass of the loop, during which the enclosed block is executed.

In [33]:
```
for letter in l:
    print(letter)
```

```
a
b
c
```

Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

For loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

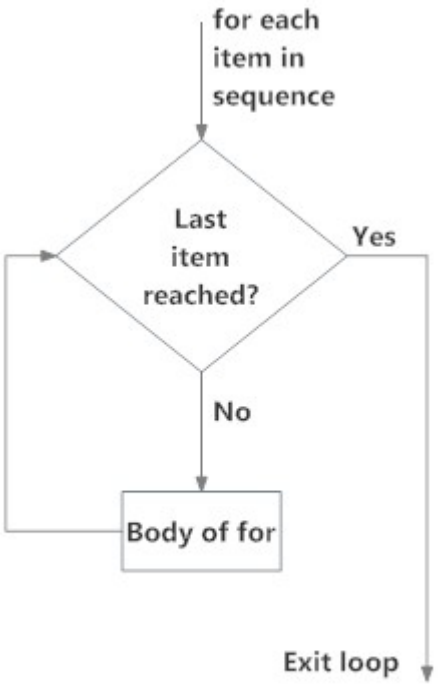


Fig: operation of for loop

In [34]:
```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

# Output: The sum is 48
print("The sum is", sum)
```

```
The sum is 48
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

break statement can be used to stop a for loop. In such case, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

In [35]:
```
digits = [0, 1, 5]

for i in digits:
    print(i)
else:
    print("No items left.")
```

```
0
1
5
No items left.
```

While loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.

In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

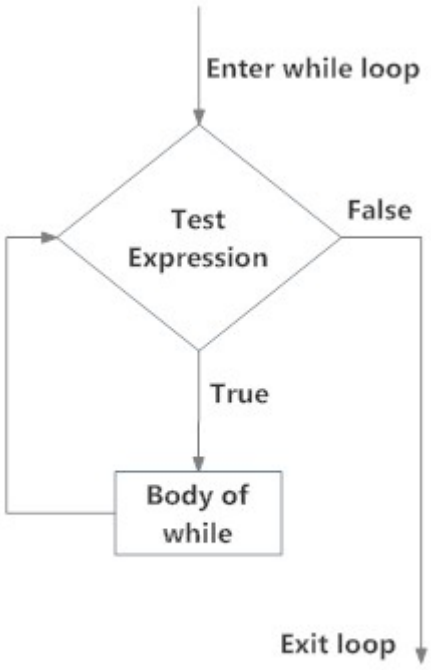


Fig: operation of while loop

In [36]:
```
# Program to add natural
# numbers upto
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

```
The sum is 55
```

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

In [37]:
```
print(range(10))
```

```
range(0, 10)
```

In [38]:
```
print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [39]:
```
print(list(range(2, 8)))
```

```
[2, 3, 4, 5, 6, 7]
```

In [40]:
```
print(list(range(2, 20, 3)))
```

```
[2, 5, 8, 11, 14, 17]
```

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

In [41]:
```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like", genre[i])
```

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

break and continue statement

In Python, break and continue statements can alter the flow of a normal loop.

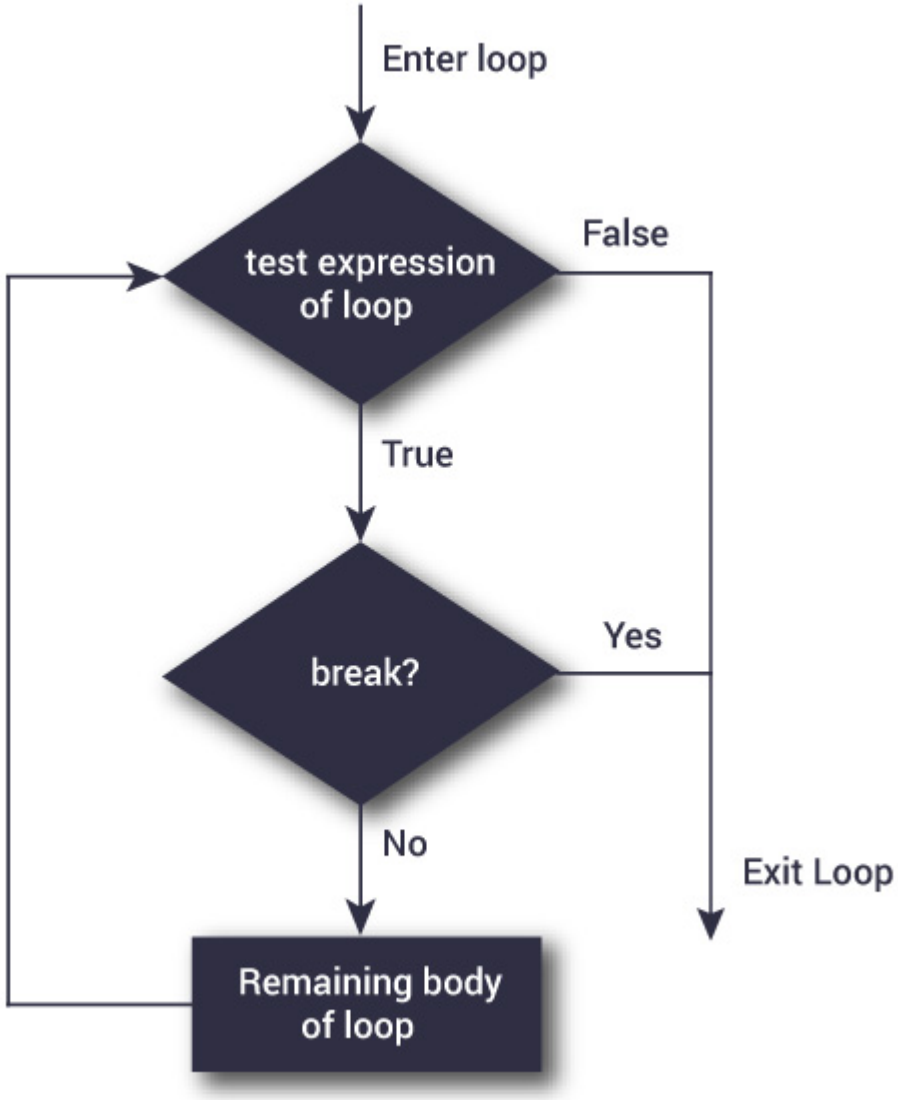
Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without cheking test expression.

The break and continue statements are used in these cases.

break

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.



In [42]:
```
# Use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

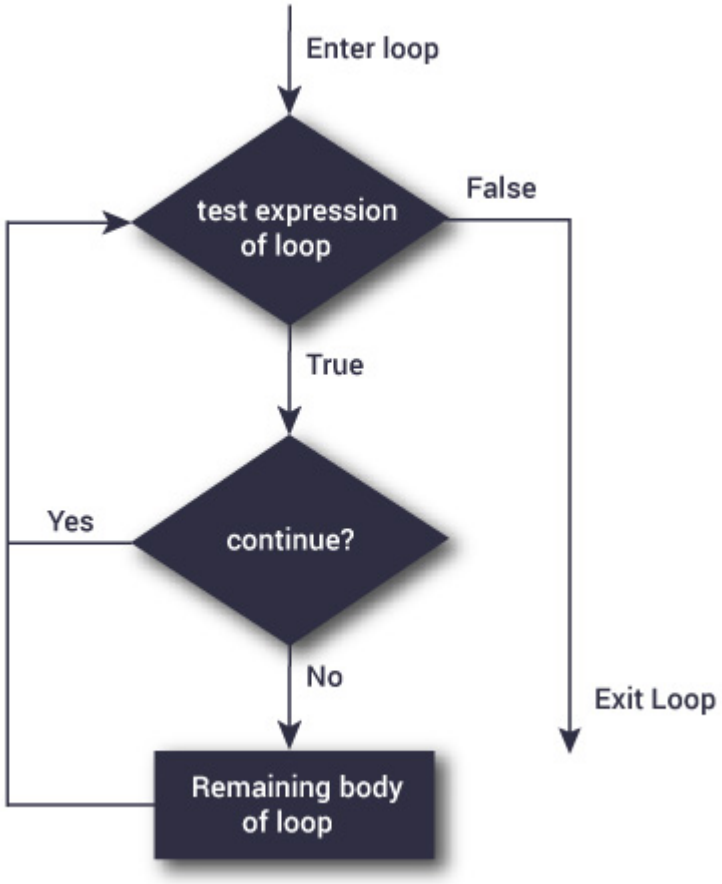
```
s
t
r
The end
```

In this program, we iterate through the "string" sequence. We check if the letter is "i", upon which we

break from the loop. Hence, we see in our output that all the letters up till "i" gets printed. After that, the loop terminates.

continue

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



In [43]:
```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

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

This program is same as the above example except the break statement has been replaced with continue.

We continue with the loop, if the string is "i", not executing the rest of the block. Hence, we see in our output that all the letters except "i" gets printed.

In [1]:
```
# Program to take the input string from the user.

name = input("What is your name? ")
type(name)
```

```
What is your name? Onkar
```

Out[1]:
```
str
```

In [2]:
```
# Program to read integers from user

age = input("What is your age? ")
print ("Your age is: ", age)
type(age)
```

```
What is your age? 25
Your age is:  25
```

Out[2]:
```
str
```

In [3]:
```
# Let's have one more example

name = input("What is your name? ")
print (" It was nice talking you " + name + "!")
age = input("Enter your age? ")
print("Hey, you are already " + age + " years old, " + name + "!")
```

```
What is your name? Satyam
 It was nice talking you Satyam!
Enter your age? 26
Hey, you are already 26 years old, Satyam!
```