Popular tools used in data science

- Data pre-processing and analysis
  - Python, R, Microsoft Excel, SAS, SPSS
- Data exploration and visualization
  - Tableau, Qlikview, Microsoft Excel
- Parallel and distributed computing incase of big data
  - Apache Spark, Apache Hadoop

Evolution of Python

- Python was developed by Guido van Rossum in the late eighties at the 'National Research Institute for Mathematics and Computer Science' at Netherlands
- Python Editions
  - Python 1.0
  - Python 2.0
  - Python 3.0

Python as a programming language

- Supports multiple programming paradigm
  - Functional, Structural, OOPs, etc.
- Dynamic typing
  - Runtime type safety checks
- Reference counts
  - Deallocates objects which are not used for long
- Late binding
  - Methods are looked up by name during runtime
- Python's design is guided by 20 aphorisms as described in Zen of Python by Tim Peters

Python as a programming language

- Standard CPython interpreter is managed by "Python Software Foundation"
- There are other interpreters namely JPython (Java), Iron Python (C#), Stackless Python (C, used for parallelism), PyPy (Python itself JIT compilation)
- Standard libraries are written in python itself
- High standards of readability

Python as a programming language

- Cross-platform (Windows, Linux, Mac)
- Highly supported by a large community group
- Better error handle

Python as a programming language

- Comparison to Java
- Python vs Java
  - Java is statically typed i.e. type safety is checked during compilation (static compilation)
  - Thus in Java the time required to develop the code is more
  - Python which is dynamically typed compensates for huge compilation time when compared to Java
  - Codes which are dynamically typed tend to be less verbose therefore offering more readability

Advantages of using python

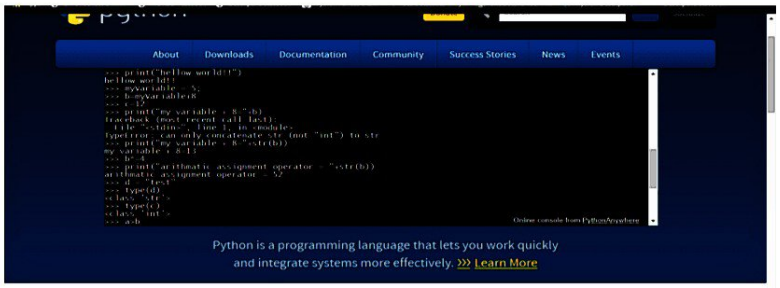
- Python has several features that make it well suited for data science
- Open source and community development
  - Developed under Open Source Initiative license making it free to use and distribute even commercially
- Syntax used is simple to understand and code
- Libraries designed for specific data science tasks
- Combines well with majority of the cloud platform service providers

Coding environment

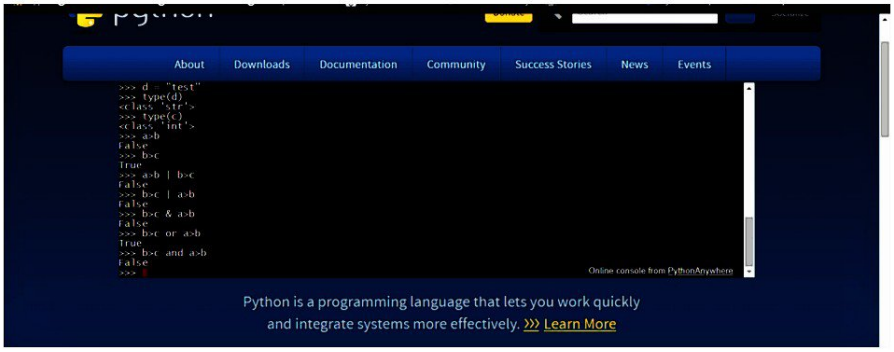
- A software program can be written using a terminal, a command prompt (cmd), a text editor or through an Integrated Development Environment (IDE)
- The program needs to be saved in a file with an appropriate extension (.py for python, .mat for matlab, etc...) and can be executed in corresponding environment (Python, Matlab, etc…)
- Integrated Development Environment (IDE) is a software product solely developed to support software development in various or specific programming language(s)

Coding environment

- Python 2.x support will be available till 2020
- Python 3.x is an enhanced version of 2.x and will only be maintained from 3.6.x post 2020
- Install basic python version or use the online python console as in https://www.python.org/
- Execute following commands and view the outputs in terminal or command prompt
  - Basic print statement
  - Naming conventions for variables and functions, operators
  - Conditional operations, looping statements (nested)
  - Function declaration and calling
  - Installing modules



https://www.python.org/



https://www.python.org/

Integrated development environment (IDE)

- Software application consisting of a cohesive unit of tools required for development
- Designed to simplify software development
- Utilities provided by IDEs include tools for managing, compiling, deploying and debugging software

Coding environment- IDE

- An IDE usually comprises of
  - Source code editor
  - Compiler
  - Debugger
  - Additional features include syntax and error highlighting, code completion
- Offers supports in building and executing the program along with debugging the code from within the environment

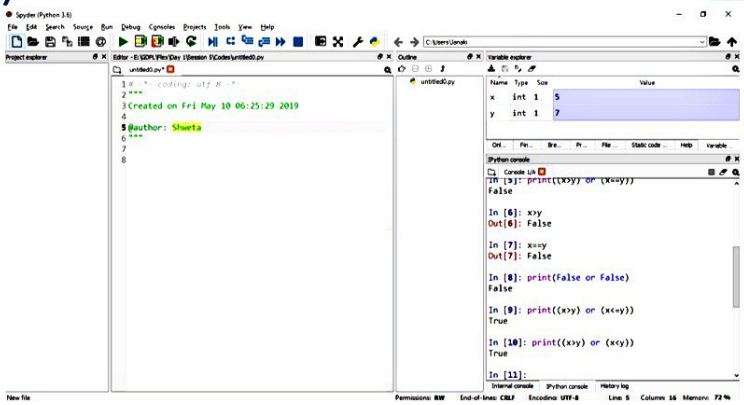
Coding environment- IDE

- Best IDEs provide version control features
- Eclipse+PyDev, SublimeText, Atom, GNU Emacs, Vi/Vim, Visual Studio, Visual Studio Code are general IDEs with python support
- Apart from these some of the python specific editors include Pycharm, Jupyter, Spyder, Thonny

Spyder

- Supported across Linux, Mac OS X and Windows platforms
- Available as open source version
- Can be installed separately or through Anaconda distribution
- Developed for Python and specifically data science
- Features include
  - Code editor with robust syntax and error highlighting
  - Code completion and navigation
  - Debugger
  - Integrated document
- Interface similar to MATLAB and RStudio

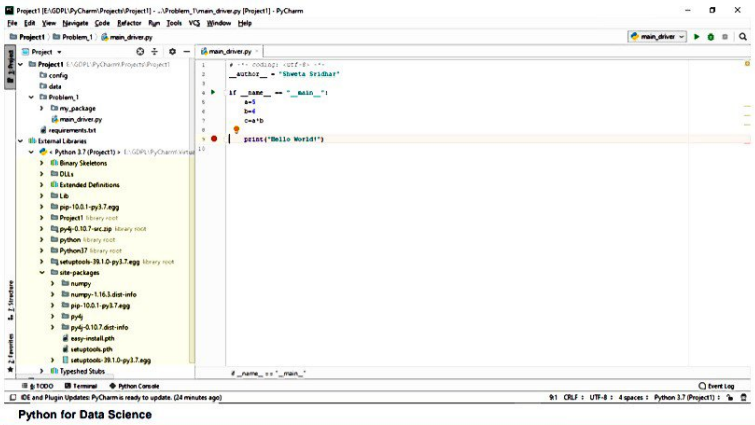
Spyder



PyCharm

- Supported across Linux, Mac OS X and Windows platforms
- Available as community (free open source) and professional (paid) version
- Supports only Python
- Can be installed separately or through Anaconda distribution
- Features include
  - Code editor provides syntax and error highlighting
  - Code completion and navigation
  - Unit testing
  - Debugger
  - Version control

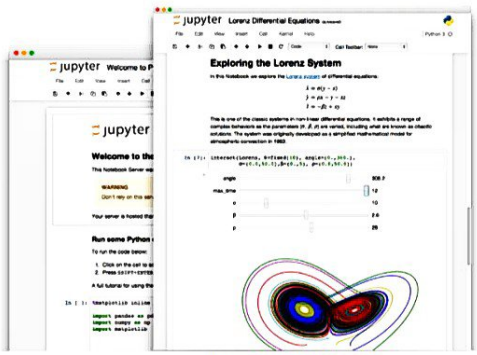
PyCharm



Jupyter Notebook

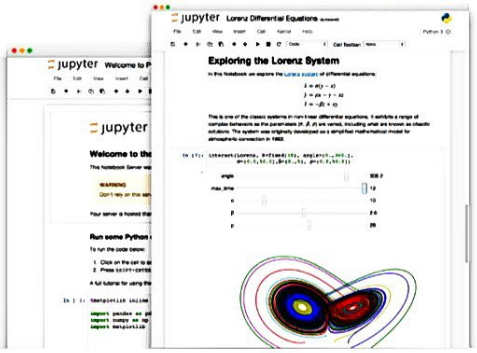
- Web application that allows creation and manipulation of documents called 'notebook'
- Supported across Linux, Mac OS X and Windows platforms
- Available as open source version

Jupyter Notebook



Source-https://jupyter.org/

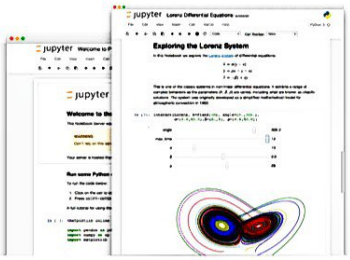
Jupyter Notebook



Source-https://jupyter.org/

Jupyter Notebook

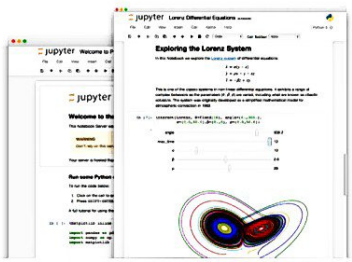
- Bundled with Anaconda distribution or can be installed separately
- Supports Julia, Python, R and Scala
- Consists of ordered collection of input and output cells that contain code, text, plots etc.



Source-https://jupyter.org/

Jupyter Notebook

- Allows sharing of code and narrative text through output formats like PDF, HTML etc.
  - Education and presentation tool
- Lacks most of the features of a good IDE

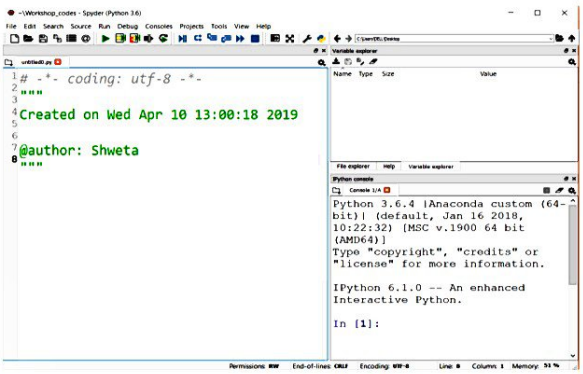


Source-https://jupyter.org/

How to choose the best IDE?

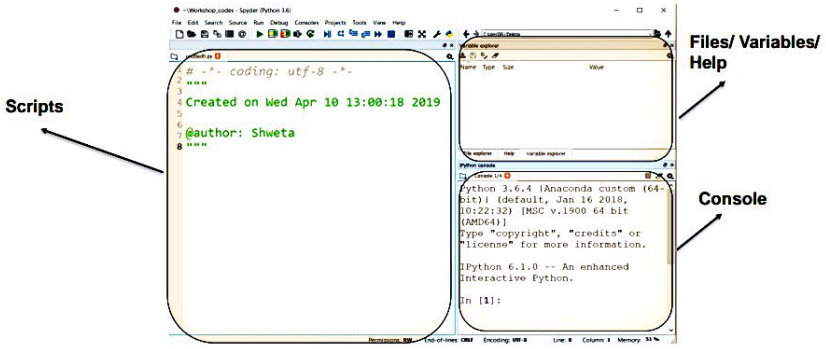
- Requirements
- Working with different IDEs helps us understand our own requirement

Appearance of Spyder



Python version 3.6

Appearance of Spyder



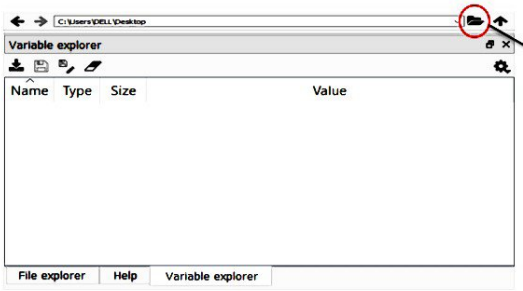
Setting working directory

Setting working directory

- There are three ways to set a working directory
  - Icon
  - Using library ***os***
  - Using command ***cd***

Setting working directory

Method 1



To choose a working directory, click on the icon

Choose a suitable location by clicking on the indicated icon

Setting working directory

- Type the following in the console

Method 2

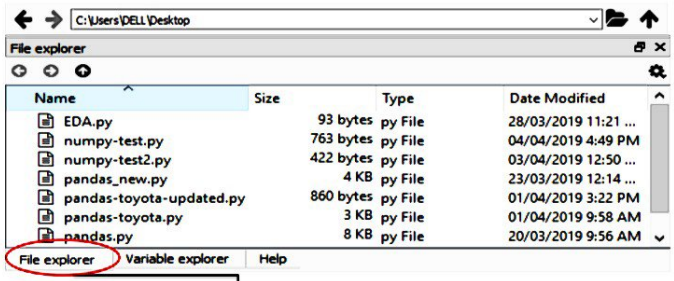
```
# Import os to setup the working directory
import os

# Setting the working the directory
os.chdir('C:/Users/DELL/Desktop')
```

Method 3

```
cd C:/Users/DELL/Desktop
```


Accessing file explorer



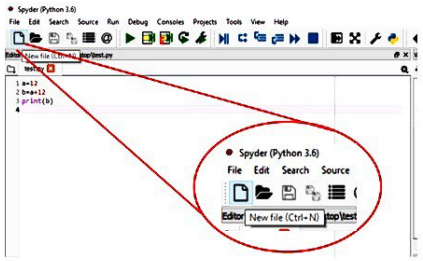
Click here to check for files after setting the working directory

File creation

Creating a script file

- There are two ways of creating a script file
- By clicking the icon " " below the menubar

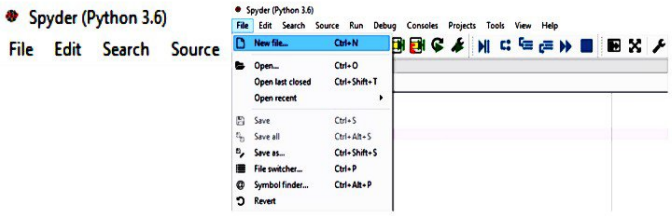
Method 1



Creating a script file

- By clicking the "File" menu in the menubar and select "New File"

Method 2



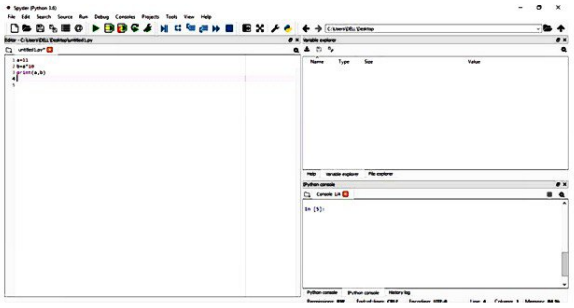
Variable

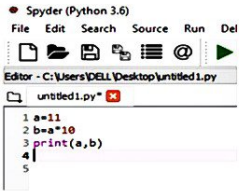
13-15 / 21

Variable

- An identifier containing a known information
- Information is referred to as value
- Variable name points to a memory address or a storage location and used to reference the stored value

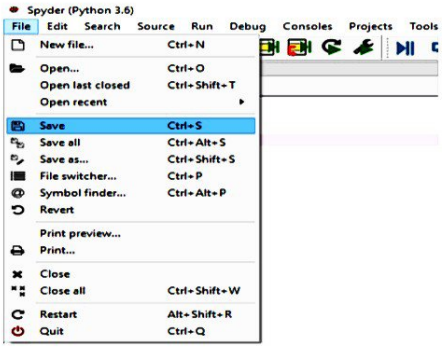
Creating variables



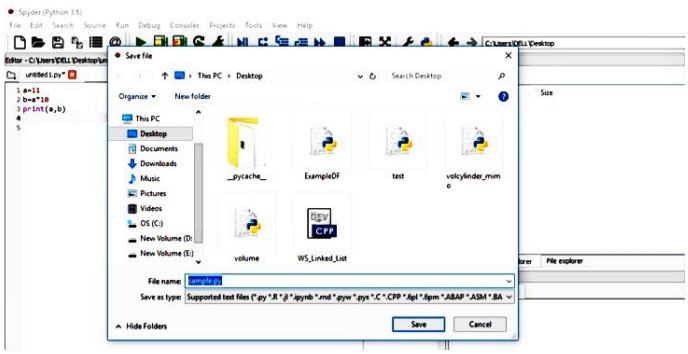


Saving script files

Saving a script file

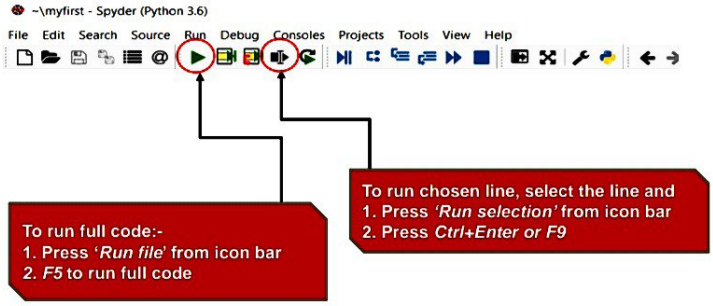


Saving a script file for the first time

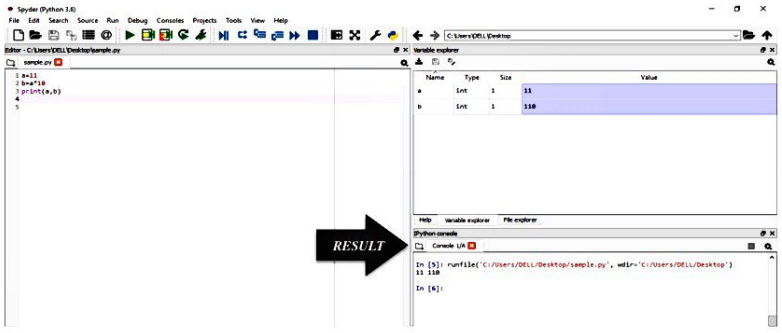


File execution

Executing script files

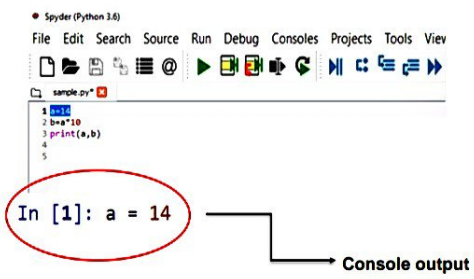


Executing script files using Run file/F5



Executing script files using Run selection/F9

Step 1: Assign a new value of 14 to 'a' in the script and press F9



Executing script files using Run selection/F9

Step 2: Select line 2 and press F9

```
In [2]: b = a*10
```
→ Console output

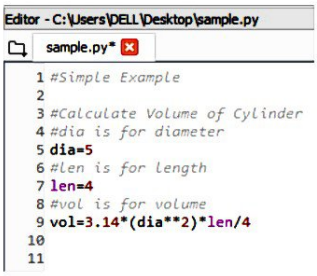
Step 3: Select line 3 and press F9

```
In [4]: print(a,b)
14 140
```
→ Console output

Commenting script files

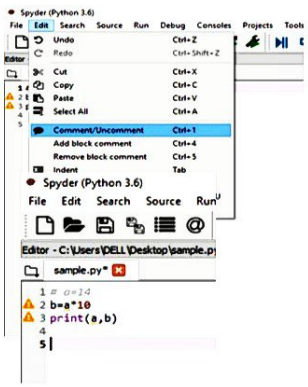
Commenting lines of codes

- Adding comments will help in understanding algorithms used while developing codes
- In practice, commented statements will be added before the code and begin with a '#'
- Multiple lines can also be commented



Commenting multiple lines

- Select lines that have to be commented and then press "Ctrl + 1"
- Select "Edit" in menu and select "Comment/Uncomment"
- Uses - to add description, render lines of code inert during testing



GITAA

Clearing console and environment

GITAA

Clearing an overpopulated console

Console

IPython console
Console 1/A

```
In [5]: a=14

In [6]: b=a*10

In [7]: print(a,b)
14 140
```

Type *%clear* in console

IPython console
Console 1/A

```
In [5]: a=14

In [6]: b=a*10

In [7]: print(a,b)
14 140

In [8]: %clear
```

Place cursor on console and press *Ctrl+L*

IPython console
Console 1/A

```
In [5]: a=14

In [6]: b=a*10

In [7]: print(a,b)
14 140

In [8]:
```


GITAA

After clearing an overpopulated console

Variable explorer

Name	Type	Size	Value
a	int	1	14
b	int	1	140

File explorer | Variable explorer | Help

IPython console
Console 1/A

```
In [9]:
```


GITAA

Removing/deleting variable(s)

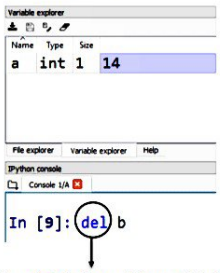
Environment

Variable explorer

Name	Type	Size	Value
a	int	1	14
b	int	1	140

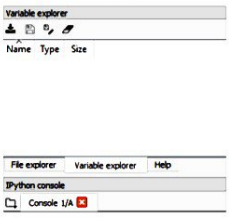
Removing/deleting variable(s)

Removing single variable



Using **del** followed by variable name

Removing multiple variables



Clearing the entire environment at once

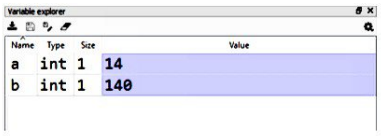
- There are two ways to clear the environment

Method 1

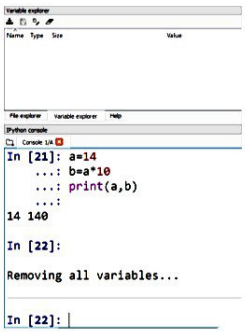
Type **%reset** in console and type **'y'** after the prompt

Clearing the entire environment at once

Method 2



Click the  symbol to remove variables in environment

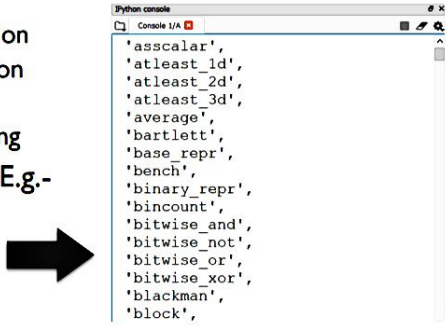


Basic libraries in Python

Basic libraries in Python

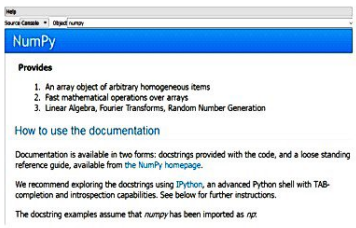
- Basic libraries
  - NumPy – Numerical Python
  - Pandas – Dataframe Python
  - Matplotlib - Visualization
  - Sklearn – Machine Learning
- Modules within a library. E.g.-

```
import numpy
content  = dir(numpy)
print(content)
```



Help in Python

Type the name of the library in 'Object'



The following are the sub libraries

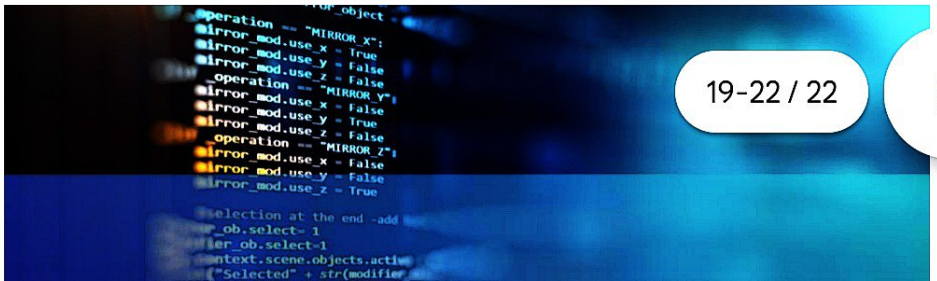
Available subpackages

doc
Topical documentation on broadcasting, indexing, etc.
lib
Basic functions used by several sub-packages.
random
Core Random Tools
linalg
Core Linear Algebra Tools
fft
Core FFT routines
polynomial
Polynomial tools
testing
NumPy testing tools
f2py
Fortran to Python Interface Generator.
distutils
Enhancements to distutils with support for Fortran compilers support and more.

Note: You can click the details of the sublibraries by typing ***libraryname.sublibraryname*** under object
Eg- **numpy.lib** in object

Summary

- Execute Python script file
- Commenting lines of code
- Clearing console and environment
- Basic libraries in Python



Naming variables

- Values assigned to variables using an assignment operator '='
- Variable name should be short and descriptive
  - Avoid using variable names that clash with inbuilt functions
- Designed to indicate the intent of its use to the end user
- Avoid one character variable names
  - One character variable names are usually used in looping constructs, functions, etc

Naming variables

- Variables can be named alphanumerically

```
Age=55   age=55   age2=55   Age2=55
```

- However the first letter must start with an alphabet (lowercase or uppercase)

```
In [3]: 2age=55
  File "<ipython-input-3-00efac86f1ae>", line 1
    2age=55
       ^
SyntaxError: invalid syntax
```


Naming variables

- Other special character
  - Underscore (_) → `Employee_id=501`
  - Use of any other special character will throw an error →

```
In [6]: Employee@id=501
  File "<ipython-input-6-e2ec8cb31ebe>", line 1
    Employee@id=501
                ^
SyntaxError: can't assign to operator
```

  - Variable names should not begin or end with underscore even though both are allowed →

```
_age=55
age_=55
```


Naming conventions

- Commonly accepted case types
  - Camel (lower and upper)

```
ageEmp=45      AgeEmp=45
```

  - Snake

```
age_emp=45     Age_emp=45
```

  - Pascal

```
AgeEmp=45
```

Assigning values to multiple variables

Code

```
Physics,Chemistry,Mathematics=89,90,75
```

Values reflected in environment

Variable explorer

Name	Type	Size	Value
Chemistry	int	1	90
Mathematics	int	1	75
Physics	int	1	89

Data types

Basic data types

Basic data types	Description	Values	Representation
Boolean	represents two values of logic and associated with conditional statements	True and False	bool
Integer	positive and negative whole numbers	set of all integers, Z	int
Complex	contains real and imaginary part (a+ib)	set of complex numbers	complex
Float	real numbers	floating point numbers	float
String	all strings or characters enclosed between single or double quotes	sequence of characters	str

Identifying object data type

- Find data type of object using
- Syntax: `type(object)`

```
Employee_name="Ram"
Age=55
Height=150.6
```

Checking the data type of an object

```
In [10]: type(Employee_name)
Out[10]: str

In [11]: type(Age)
Out[11]: int

In [12]: type(Height)
Out[12]: float
```

Verifying object data type

- Verify if an object is of a certain data type
- Syntax: `type(object) is datatype`

```
Employee_name="Ram"
Age=55
Height=150.6
```

Verifying the data type of an object

```
In [13]: type(Height) is int
Out[13]: False

In [14]: type(Age) is float
Out[14]: False

In [15]: type(Employee_name) is str
Out[15]: True
```


Coercing object to new data type

- Convert the data type of an object to another
- Syntax: `datatype(object)`
- Changes can be stored in same variable or in different variable

```
Employee_name="Ram"
Age=55
Height=150.6
```

Coercing the data type of an object

```
In [16]: type(Height)
Out[16]: float

In [17]: ht=int(Height)

In [18]: type(ht)
Out[18]: int

In [19]: Height=int(Height)

In [20]: type(Height)
Out[20]: int
```


Coercing object to new data type

- Only few coercions are accepted
- Consider the variable '*Salary_tier*' which is of string data type
- '*Salary_tier*' contains an integer enclosed between single quotes

```
Salary_tier='1'
```

Coercing the data type of an object

```
In [20]: type(Salary_tier)
Out[20]: str

In [21]:Salary_tier=int(Salary_tier)

In [24]: type(Salary_tier)
Out[24]: int
```


Coercing object to new data type

- However if the value enclosed within the quotes is a string then conversions will not be possible

```
In [19]: Employee_name="Ram"

In [20]: Employee_name=float(Employee_name)
Traceback (most recent call last):

  File "<ipython-input-20-b0286eb77de0>", line 1, in <module>
    Employee_name=float(Employee_name)

ValueError: could not convert string to float: 'Ram'
```


Summary

Operators and operands

- Operators are special symbols that help in carrying out an assignment operation or arithmetic or logical computation
- Value that the operator operates on is called operand

```
In [1]: 2+3
Out[1]: 5
```


Arithmetic operators

- Used to perform mathematical operations between two operands
- Create two variable a and b with values 10 and 5 respectively

```
a=10    b=5
```

Symbol	Operation	Example
+	Addition	`In [3]: a+b` `Out[3]: 15`

Arithmetic operators

- Used to perform mathematical operations between two operands
- Create two variable a and b with values 10 and 5 respectively

```
a=10    b=5
```

Symbol	Operation	Example
+	Addition	`In [3]: a+b` `Out[3]: 15`
-	Subtraction	`In [4]: a-b` `Out[4]: 5`

Arithmetic operators

Symbol	Operation	Example
*	Multiplication	`In [5]: a*b` `Out[5]: 50`

Arithmetic operators

Symbol	Operation	Example
*	Multiplication	In [5]: a*b Out[5]: 50
/	Division	In [6]: a/b Out[6]: 2.0

Arithmetic operators

Symbol	Operation	Example
*	Multiplication	In [5]: a*b Out[5]: 50
/	Division	In [6]: a/b Out[6]: 2.0
%	Remainder	In [8]: a%b Out[8]: 0

Arithmetic operators

Symbol	Operation	Example
*	Multiplication	In [5]: a*b Out[5]: 50
/	Division	In [6]: a/b Out[6]: 2.0
%	Remainder	In [8]: a%b Out[8]: 0
**	Exponent	In [7]: a**b Out[7]: 100000

Hierarchy of arithmetic operators

Decreasing order of precedence	Operation
Parentheses	( )
Exponent	**
Division	/
Multiplication	*
Addition and subtraction	+,-

$$A = 7 - 2 \times \frac{27}{3^2} + 4$$

```
In [10]: A=7-2*(27/3**2)+4

In [11]: print(A)
5.0
```

Assignment operators

- Used to assign values to variables

Symbol	Operation	Example
=	Assign values from right side operands to left side operand	a=10 b=5 .

Assignment operators

- Used to assign values to variables

Symbol	Operation	Example
=	Assign values from right side operands to left side operand	a=10 b=5 .
+=	Adds right operand to left operand and stores result on left side operand (a=a+b)	In [55]: a+=b ...: print(a) 15

Assignment operators

- Used to assign values to variables

Symbol	Operation	Example
=	Assign values from right side operands to left side operand	a=10 b=5 .
+=	Adds right operand to left operand and stores result on left side operand (a=a+b)	In [55]: a+=b ...: print(a) 15
-=	Subtracts right operand from left operand and stores result on left side operand (a=a-b)	In [57]: a-=b ...: print(a) 5

Assignment operators

Symbol	Operation	Example
*=	Multiplies right operand from left operand and stores result on left side operand (a=a*b)	In [59]: a*=b ...: print(a) 50

Assignment operators

Symbol	Operation	Example

*=	Multiplies right operand from left operand and stores result on left side operand (a=a*b)	`In [59]: a*=b`
		`...: print(a)`
		`50`

/=	Divides right operand from left operand and stores result on left side operand (a=a/b)	`In [62]: a/=b`
		`...: print(a)`
		`2.0`

Relational or comparison operators

- Tests numerical equalities and inequalities between two operands and returns a boolean value
- All operators have same precedence
- Create two variables x and y with values 5 and 7 respectively

```
In [1]: x = 5
In [2]: y = 7
```

Symbol	Operation	Example
<	Strictly less than	`In [3]: print(x<y)` `True`

Relational or comparison operators

- Tests numerical equalities and inequalities between two operands and returns a boolean value
- All operators have same precedence
- Create two variables x and y with values 5 and 7 respectively

```
In [1]: x = 5
In [2]: y = 7
```

Symbol	Operation	Example
<	Strictly less than	`In [3]: print(x<y)` `True`
<=	Less than equal to	`In [4]: print(x<=y)` `True`

Relational or comparison operators

Symbol	Operation	Example
>	Strictly greater than	`In [5]: print(x>y)` `False`
>=	Greater than equal to	`In [6]: print(x>=y)` `False`

Relational or comparison operators

Symbol	Operation	Example
>	Strictly greater than	`In [5]: print(x>y)` `False`
>=	Greater than equal to	`In [6]: print(x>=y)` `False`
==	Equal to equal to	`In [7]: print(x==y)` `False`

Relational or comparison operators

Symbol	Operation	Example
>	Strictly greater than	`In [5]: print(x>y)` `False`
>=	Greater than equal to	`In [6]: print(x>=y)` `False`
==	Equal to equal to	`In [7]: print(x==y)` `False`
!=	Not equal to	`In [8]: print(x!=y)` `True`

Logical operators

- Used when operands are conditional statements and returns boolean value
- In python, logical operators are designed to work with scalars or boolean values

Symbol	Operation	Example
or	Logical OR	`In [9] print((x>y) or (x<y))` `True`

Logical operators

- Used when operands are conditional statements and returns boolean value
- In python, logical operators are designed to work with scalars or boolean values

Symbol	Operation	Example
or	Logical OR	`In [9] print((x>y) or (x<y))` `True`
and	Logical AND	`In [10]: print((x>y) and (x<y))` `False`

Logical operators

- Used when operands are conditional statements and returns boolean value
- In python, logical operators are designed to work with scalars or boolean values

Symbol	Operation	Example
or	Logical OR	`In [9] print((x>y) or (x<y))` `True`
and	Logical AND	`In [10]: print((x>y) and (x<y))` `False`
not	Logical NOT	`In [11]: print(not (x==y))` `True`

Bitwise operators

- Used when operands are integers
- Integers are treated as a string of binary digits
- Operates bit by bit
- Can also operate on conditional statements which compare scalar values or arrays
- Bitwise OR (|), AND(&)

Bitwise operators

- Create two variables x and y with values 5 and 7 respectively

```
In [1]: x = 5
In [2]: y = 7
```

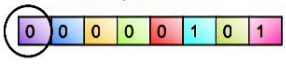
- Binary code for 5 is ***0000 0101*** and for 7 is ***0000 0111***
- ***0*** corresponds to ***False*** and ***1*** corresponds to ***True***
- In bitwise OR (|), operator copies a bit to the result if it exists in either operand
- In bitwise AND (&), operator copies a bit to the result if it exists in both operands

Bitwise OR on integers

Code and output in console

```
In [47]: x | y
Out[47]: 7
```

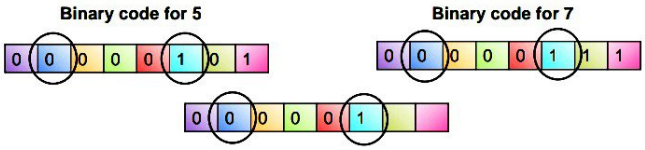
Binary code for 5



Binary code for 7

- 0 present in corresponding positions, therefore resultant cell is also 0

Bitwise OR on integers

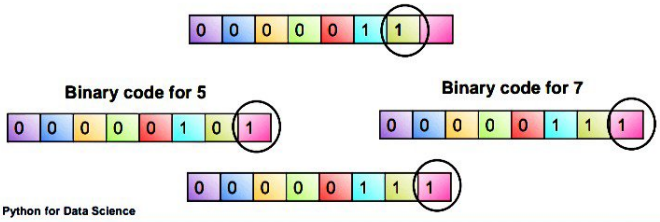


- 0 present in positions 2-5, therefore resultant cell will also contain 0
- In the 6th position, 1 is present in both operands and hence resultant will also contain 1

Bitwise OR on integers

- The 7th position has 0 in the first operand and 1 in the second

- Since this is an OR operator, only the True condition is considered



Bitwise operators

- Bitwise operators can also operate on conditional statements

Symbol	Operation	Example
|	Bitwise OR	`In [9]: print((x<y)|(x==y))` `True`

Bitwise operators

- Bitwise operators can also operate on conditional statements

Symbol	Operation	Example
|	Bitwise OR	`In [9]: print((x<y)|(x==y))` `True`
&	Bitwise AND	`In [10]: print((x<y)&(x==y))` `False`

|	Bitwise OR	In [9]: print((x<y)|(x==y)) True

Bitwise operators

- Bitwise operators can also operate on conditional statements

Symbol	Operation	Example
|	Bitwise OR	In [9]: print((x<y)|(x==y)) True
&	Bitwise AND	In [10]: print((x<y)&(x==y)) False

Precedence of operators

Decreasing order of precedence	Operation
Parentheses	( )
Exponent	**
Division	/
Multiplication	*
Addition and subtraction	+,-
Bitwise AND	&

Precedence of operators

Decreasing order of precedence	Operation
Bitwise OR	|
Relational/ comparison operators	==, !=, >, >=, <, <=
Logical NOT	not
Logical AND	and
Logical OR	or

Summary

- Important operators
  - Arithmetic
  - Assignment
  - Relational
  - Logical
  - Bitwise

Operators

Unary Operators

- +, -, , /, //, * and % are known as operators and these operators can be unary or binary
- A unary operator has only one operand
- The unary - (minus) operator yields the negation of its numeric argument
- The unary + (plus) operator yields its numeric argument unchanged
- The unary ~ (invert) operator yields the bit-wise inversion of its plain or long integer argument

Example

The - (minus) operator is used to negate any positive number

In [1]:

```
-15      # in this case the - (minus) operator is acting as a unary operator
```

Out[1]:

```
-15
```

In [2]:

```
100 - 40 # The -(minus) operator is acting as a binary operator
```

Out[2]:

```
60
```

Identity operators

- Identity operators are used to compare if two objects are same with the same memory location
- They are usually used to determine the data type of a variable
- The identity operators are 'is' and 'is not'

'is' operator - evaluates to true if the variables on either side of the operator point to the same object and false otherwise

In [3]:

```
a = 15
if (type(a) is float):
    print ("true")
else:
    print ("false") # returns false because the data type of 'a' is 'int' not 'float'
```

```
false
```

'is not' operator - evaluates to false if the variables on either side of the operator point to the same object and true otherwise

In [4]:

```
b = 15.6
if (type(b) is not float):
    print ("true")
else:
    print ("false") # returns false because the data type of 'b' is 'float' not 'int'
```

```
false
```

Membership Operators

- Membership operators are operators used to validate the membership of a value in a sequence
- The membership operators are 'in' and 'not in'

'in' operator - checks if a value exists in a sequence or not

It evaluates to true if it finds a variable in the specified sequence and false otherwise

In [5]:

Membership Operators

- Membership operators are operators used to validate the membership of a value in a sequence
- The membership operators are 'in' and 'not in'

'in' operator - checks if a value exists in a sequence or not

It evaluates to true if it finds a variable in the specified sequence and false otherwise

In [5]:

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

print(4 in x) # returns True because 4 exists in the x
```

```
True
```

'not in' operator - evaluates to true if it does not find a variable in the specified sequence and false otherwise

In [6]:

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

print(8 not in y) # returns True because 8 doesn't exists in the y
```

```
True
```


Bitwise Operators

Operator	Name	Description	Syntax
&	Bitwise AND	Sets each bit to 1 if both bits are 1	x & y
|	Bitwise OR	Sets each bit to 1 if one of two bits is 1	x | y
~	Bitwise NOT	Inverts all the bits	~x
^	Bitwise XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
>>	Bitwise right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x>>
<<	Bitwise left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x<<

END OF SCRIPT