# Data Analysis Using Python

Samatrix Consulting Pvt Ltd

Samatrix.io

# Python Programming Basics

# Ipython Interpreter

- Python is an interpreted language that executes one statement one at a time. You can invoke the standard interactive Python interpreter on the command line with the python command

```
$ python
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> a = 7
>>> print(a)
7
```

# Ipython Interpreter

In this example you can see `>>>` prompt where you can type the code expressions. You can exit the Python interpreter and return to the command prompt by typing `exit()` or pressing `Ctrl-D`

You can easily run `.py` files. Suppose you have a file `my_first_program.py`. the content of the file is

```
print('Hello World')
```

# Ipython Interpreter

You can run the file by executing the following command. However, the file `my_first_program.py` must be in the current working terminal directory.

```
$ python my_first_program.py
Hello world
```

Majority of data scientists prefer IPython, an enhanced Python interpreter, or Jupyter notebooks, web-based code notebooks. If you are using IPython or Jupyter notebooks, you can use `%run` command to run `.py` files

# Ipython Interpreter

```
$ ipython
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?'
for help.

In [1]: %run my_first_program.py
Hello World

In [2]:
```

In this case the IPython prompt is numbered `In [2]:` style compared with the standard `>>>` prompt

# Tab Completion

- One of the major differences between standard Python shell and IPython shell is tab completion.
- When you enter an expression in the shell and press the Tab key, IPython shell will search the namespace of all the variables (objects, functions, etc.) that would match the characters that you have typed so far and show them in a drop down box.

# Tab Completion

```
$ ipython
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for
help.

In [1]: an_orange = 30

In [2]: an_animal = 29

In [3]: an
            an_animal
            an_orange
```

Samatrix.io

# Tab Completion

- In the example given above, the IPython displayed both the variables that we declared and built-in function any.
- You can also complete the methods and attributes on any object after typing a period

```
In [3]: a_list = [1, 2, 3, 4]

In [4]: a_list.
                append()   count()    insert()   reverse()
                clear()    extend()   pop()      sort()
                copy()     index()    remove()
```

# Tab Completion

- You can also use the tab completion for modules

```
In [4]: import datetime

In [5]: datetime.
                    date            MAXYEAR         time            tzinfo
                    datetime        MINYEAR         timedelta
                    datetime_CAPI   sys             timezone
```

# Introspection

A question mark (?) before or after a variable will display some general information about the object:

```
In [5]: b = [1, 2, 3]

In [6]: b?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

# Introspection

```
In [7]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:  builtin_function_or_method
```

# Introspection

This is known as **object introspection**. Suppose we define a new function

```
In [8]: def number_add(x ,y):
   ...:     """
   ...:     Add Two Numbers and
   ...:     Return
   ...:     the sum of two arguements
   ...:     """
   ...:     return x + y
   ...:
```

# Introspection

Single question mark (?) will show the docstring:

```
In [9]: number_add?
Signature: number_add(x, y)
Docstring:
Add Two Numbers and
Return
the sum of two arguements
File:      ~/Desktop/<ipython-input-8-3594473669bf>
Type:      function
```

# Introspection

Double question mark (??) will show the function's source code:

```
In [10]: number_add??
Signature: number_add(x, y)
Source:
def number_add(x ,y):
    """
    Add Two Numbers and
    Return
    the sum of two arguements
    """
    return x + y
File:      ~/Desktop/<ipython-input-8-3594473669bf>
Type:      function
```

# Introspection

Single question mark (`?`) combined with the wildcard (`*`) will show all names matching the wildcard expression.

For example, we can search all functions in the top-level NumPy namespace that contains load:

```
In [11]: import numpy as np

In [12]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
```

# %run command

To run any file as a python program within IPython session, you can use %run command. Suppose we need to run the following `test_script.py`

```
def f(x, y, z):
        return (x + y) / z
a = 5
b = 6
c = 7.5
result = f(a, b, c)
```

we use `%run` to execute it as follows

```
In [13]: %run test_script.py
```

# %run command

In this case, all of the variables (imports, functions, and globals) that we have defined in the file, until an exception is raised, can be accessible in the IPython shell.

```
In [14]: c
Out[14]: 7.5

In [15]: result
Out[15]: 1.4666666666666666
```

# %run command

To import the script into the IPython shell, you can use `%load` command

```
In [16]: load test_script.py

In [17]: # %load test_script.py
    ...: def f(x, y, z):
    ...:     return (x + y) / z
    ...: a = 5
    ...: b = 6
    ...: c = 7.5
    ...: result = f(a, b, c)
```

# %run command

- You can interrupt any code by pressing `Ctrl-C` that will raise the `KeyboardInterrupt`.
- This will stop all the Python programs immediately except in certain unusual cases.

# Magic command

Magic commamds are IPython's special commands. These commands are not built into Python. They help you complete common tasks and control the IPython environment. To execute a magic command, you need to prefix it by the percent symbol `%`.

For example, you can use `%timeit` magic function to check the execution time of any Python statement

```
In [18]: d = np.random.randn(100, 100)
```

```
In [19]: %timeit np.dot(d, d)
48.7 µs ± 9.72 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

# Magic command

If no other variable is defined with the same name as the magic function in question, you can use the magic function by default without the percent sign.

This feature is called **automagic** and can be enabled or disabled with `%automagic`.

```
In [20]: %automagic

Automagic is OFF, % prefix IS needed for line magics.

In [21]: %automagic

Automagic is ON, % prefix IS NOT needed for line magics.
```

# Magic command

```
In [22]: pwd
Out[22]: '/Users/adec/Python'

In [23]: %pwd
Out[23]: '/Users/adec/Python'
```

Some magic functions behave like Python functions and their output can be assigned to a variable:

```
In [24]: coo = %pwd

In [25]: coo
Out[25]: '/Users/adec/Python'
```

# Magic command

| Command | Description |
|---|---|
| %quickref | Display the IPython Quick Reference Card |
| %magic | Display detailed documentation for all of the available magic commands |
| %debug | Enter the interactive debugger at the bottom of the last exception traceback |
| %hist | Print command input (and optionally output) history |
| %pdb | Automatically enter debugger after any exception |
| %paste | Execute preformatted Python code from clipboard |
| %cpaste | Open a special prompt for manually pasting Python code to be executed |
| %reset | Delete all variables/names defined in interactive namespace |
| %run script.py | Run a Python script inside IPython |
| %time statement | Report the execution time of a single statement |

# Matplotlib Integration

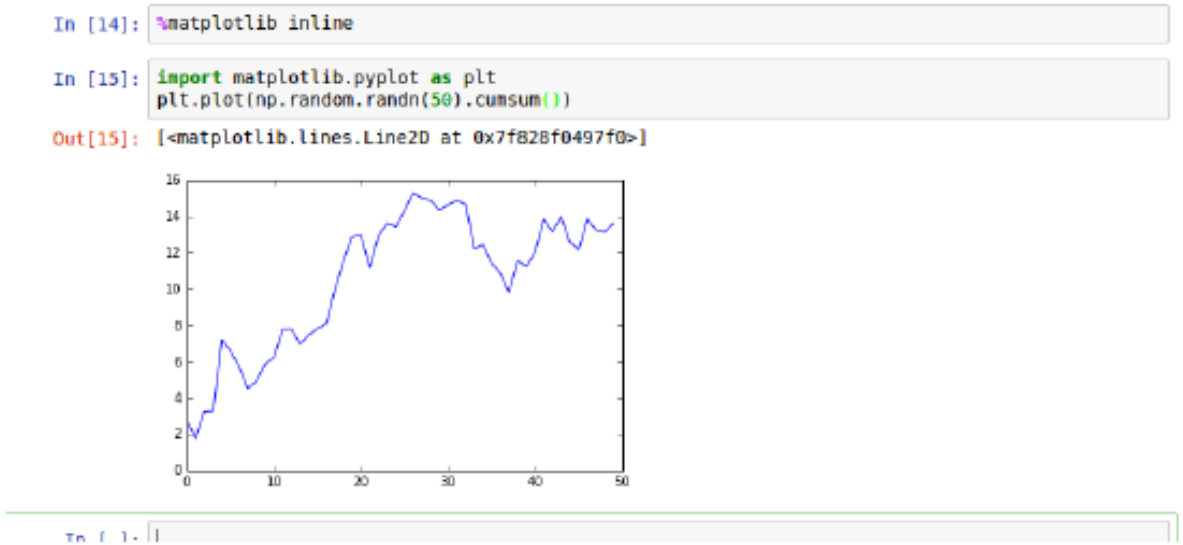
- IPython integrates well with data visualization and other user interface libraries like matplotlib.
- The %matplotlib magic function configures its integration with the IPython shell or Jupyter notebook.
- This is important, as otherwise plots you create will either not appear (notebook) or take control of the session until closed (shell).
- In the IPython shell, running %matplotlib sets up the integration so you can create multiple plot windows without interfering with the console session:

# Matplotlib Integration

```
In [26]: %matplotlib
Using matplotlib backend: MacOSX
```

In Jupyter, the command is a little different

```
In [26]: %matplotlib inline
```




Samatrix.io

# Python Programming

# Python Language Semantics

**Indentation**

To structure the code in Python, we use whitespace (tabs or spaces) instead of using braces as in the case of many other programming languages such as R, C++, Java, and Perl.

For example, the following for loop from a sorting algorithm

```
for x in array:
        if x < pivot:
                less.append(x)
        else:
                greater.append(x)
```

# Indentation

We use colon to denote the start of an indented block. After the colon, all the code should be indented by the same amount until the end of the block.

It is however recommended that we should use four spaces as default indentation

and replace tabs with four spaces

It is not required to terminate the Python statement by semicolons. You can use semicolons to separate multiple statements on a single line:

```
a = 5; b = 6; c = 7
```

It is not recommended to put multiple statements on one line as it often makes code less readable.

# Objects

- In Python, every number, string, data structure, function, class, module, and so on is referred to as a Python object.
- Each object has an associated type (e.g., string or function) and internal data.

# Comments

Python ignores any text preceded by the hash mark (pound sign) `#`. We can use the hask mark to add comments to code.

If you want to exclude certain blocks of code without deleting them, you can comment out the code:

```
outcomes = []
for li in handle:
        # for empty lines
        # if len(li) == 0:
        # continue
        outcomes.append(li.replace('foo', 'bar'))
```

You can also place comments can also occur after a line of executed code.

```
print("The comment example")  # Comment Example
```

# Variables and Assignments

When you assign a variable (or name) in Python, a reference to the object on the righthand side of the equals sign is created. For example, let's consider a list of integers:

```
In [1]: a = [1, 2, 3]
```

Now we assign the variable a to a new variable b

```
In [2]: b = a
```

In contrast to many languages that cause the data `[1, 2, 3]` to be copied as a result of the assignment, Python refers a and b to the same object, the original list `[1, 2, 3]`

# Variables and Assignments

Now if you append a, b also gets updated.

```
In [3]: a.append(4)

In [4]: b
Out[4]: [1, 2, 3, 4]
```

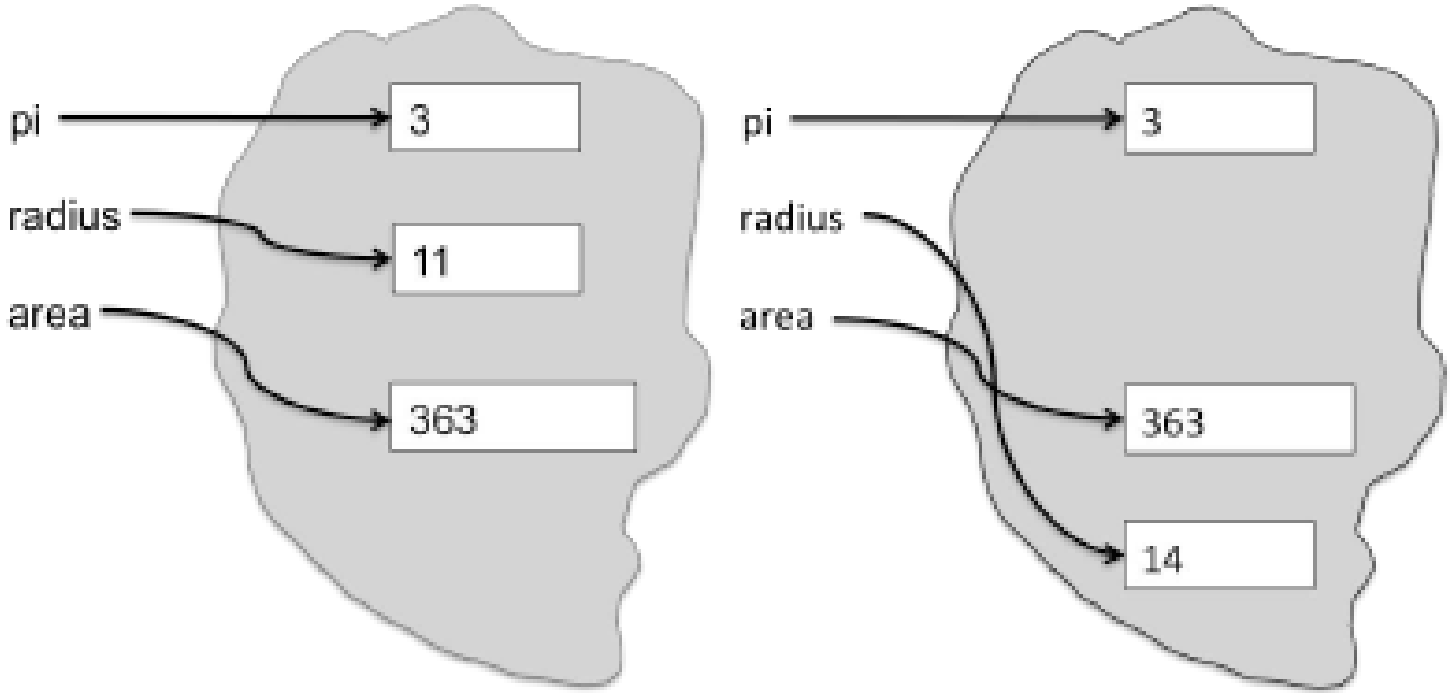
In Python, the assignment is also known as **binding**. Now consider the following code

```
pi = 3
radius = 11
area = pi * (radius**2)
radius = 14
```

# Variables and Assignments

- It first binds the names pi and radius to different objects. It then binds the name area to a third object.
- This is depicted in the left panel of Figure.
- If the program then executes radius = 11, the name radius is rebound to a different object, as shown in the right panel.
- Note that this assignment has no effect on the value to which area is bound.
- It is still bound to the object denoted by the expression 3*(11**2).

# Variables and Assignments



# Dynamic References & Strong Type

The objects in many compiled languages such as Java and C++ have type associated with them but object references in Python does not have any type associated with them. Hence, we can write as follows

```
In [5]: a = 5

In [6]: type(a)
Out[6]: int

In [7]: a = 'fruit'

In [8]: type(a)
Out[8]: str
```

# Dynamic References & Strong Type

Variables are the names of the objects and the type information is stored in the object itself. Sometime people may wrongly conclude that Python is not a “typed language” consider this example:

```
In [9]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-4dd8efb5fac1> in <module>
----> 1 '5' + 5

TypeError: can only concatenate str (not "int") to str
```

# Dynamic References & Strong Type

This example concludes that Python is a **strongly typed** language. Implicit conversion is allowed only in certain obvious instances

```
In [10]: a = 4.5
In [11]: b = 2
In [12]: type(a)
Out[12]: float
In [13]: type(b)
Out[13]: int
In [14]: c = a/b
In [15]: c
Out[15]: 2.25
In [16]: type(c)
Out[16]: float
```

# Dynamic References & Strong Type

On various occasions, it is very important to check the type of an object. You can check whether an object is an instance of a particular type using the `isinstance` function:

```
In [19]: a = 5; b = 4.5

In [20]: isinstance(a, (int,float))
Out[20]: True

In [21]: isinstance(b, (int,float))
Out[21]: True
```

# Attributes and Methods

- Objects in Python have attributes and methods
- Attributes are other Python objects that have been stored inside the object
- Methods are the functions that have been associated with an object that can have access to the object's internal data.
- Both attributes and methods can be accessed via the syntax `obj.attribute_name:`

# Attributes and Methods

```
In [22]: a = 'wind'

In [23]: a.<Press Tab>
             capitalize() encode       format       isalpha      isidentifier
             casefold     endswith     format_map   isascii      islower
             center       expandtabs   index        isdecimal    isnumeric    >
             count        find         isalnum      isdigit      isprintable
```

# Duck Typing

- On various occasions, you are not interested about the type of an object but about its methods and behaviour.
- This is known as “duck typing” after a saying “If it walks like a duck and quacks like a duck, then it’s a duck.”
- For example, you may be to know whether an object is iterable.
- It means whether it has implemented the iterator protocol.
- It also means that the object has a `__iter__` “magic method”

# Duck Typing

```
In [23]: iter('a string')
Out[23]: <str_iterator at 0x7fd6a7af40a0>

In [24]: iter(5)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-f0913ab840ef> in <module>
----> 1 iter(5)

TypeError: 'int' object is not iterable
```

# Import

A module in Python is a file with the .py extension that contains Python code.

Suppose that we had the following module:

```
# test_module.py
PI = 3.14159
def f(x):
      return x + 2
def g(a, b):
      return a + b
```

# Import

To access the variables and functions defined in `test_module.py`, from another file in the same directory we could do

```
In [25]: import test_module

In [26]: total = test_module.f(5)

In [27]: pi = test_module.PI

In [28]: from test_module import f, g, PI

In [29]: total1 = g(5, PI)
```

# Import

```
In [30]: import test_module as tm

In [31]: from test_module import PI as pi, g as gf

In [32]: r1 = tm.f(pi)

In [33]: r2 = gf(6, pi)
```

# Binary Operators and Comparison

The binary math operation and comparisons are as expected

```
In [34]: 5 - 8
Out[34]: -3

In [35]: 16 + 34.2
Out[35]: 50.2

In [36]: 5 <= 3
Out[36]: False
```

# List of Binary Operators

| Operation | Description |
|---|---|
| `a + b` | Add a and b |
| `a - b` | Subtract b from a |
| `a * b` | Multiply a by b |
| `a / b` | Divide a by b |
| `a // b` | Floor divide a by b and dropping any fractional reminder |
| `a ** b` | Raise a to the b power |
| `a & b` | True if both a and b are True |
| `a \| b` | True if either a or b is True |
| `a ^ b` | For Booleans, Ture if a or b are True but not both |
| `a == b` | True if a equals b |
| `a != b` | True if a is not equal to b |
| `a <= b, a < b` | True if a is less than (less than or equal to) b |
| `a > b, a >= b` | True if a is greater than (greater than or equal to) b |
| `a is b` | True if a and b refer to the same Python object |
| `a is not b` | True if a and b refer to different Python object |

# Binary Operators and Comparison

To check if two references refer to the same object, use the is keyword. You can use is not to check that two objects are not the same:

```
In [37]: a = [1, 2, 3]

In [38]: b = a

In [39]: c = list(a)

In [40]: a is b

Out[40]: True

In [41]: a is c

Out[41]: False

In [42]: a == c

Out[42]: True

In [43]: a is not c

Out[43]: True
```

# Binary Operators and Comparison

In this case, list copies the object a. Hence, they do not refer to the same object but their values are same. Comparing with is is not the same as the == operator

We generally use is and is not to check if the variable is None. There is only one instance of none

```
In [44]: a = None

In [45]: a is None
Out[45]: True
```

# Mutable and Immutable Objects

Several objects in Python, such as lists, dicts, NumPy arrays, most user-defined classes, are mutable.

That means we can modify the objects or the values that they contain

```
In [46]: new_list = ['book', 3, [6, 7]]

In [47]: new_list[2] = (1,2)

In [48]: new_list
Out[48]: ['book', 3, (1, 2)]
```

# Mutable and Immutable Objects

Objects such as tuples and strings immutable

```
In [49]: new_tuple = (3, 1, (1,2))

In [50]: new_tuple[1] = 4
---------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-13a3191d3765> in <module>
----> 1 new_tuple[1] = 4

TypeError: 'tuple' object does not support item assignment
```

# Scalar Types

- A scalar is a type that can have a single value such as 5, 3.14, or ‘Bob’.
- Python along with its standard library has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time.
- These “single value” types are sometimes called scalar types and we also refer to them as scalars.
- The list of standard Python scalar types has been given below

# Scalar Types

| Type | Description |
|---|---|
| `None` | The Python “null” value (only one instance of the None object exists) |
| `str` | String type; holds Unicode (UTF-8 encoded) strings |
| `bytes` | Raw ASCII bytes (or Unicode encoded as bytes) |
| `float` | Double-precision (64-bit) floating-point number |
| `bool` | A True or False value |
| `int` | Arbitrary precision signed integer |

# Numeric Type

`int` and `float` are the primary Python numeric types. An `int` can store a large value

```
In [51]: intval = 78755433
In [52]: intval ** 9
Out[52]:
116553337087250147833709029943730455749737270658869603234951056719184553
```

Python `float` type represent floating point numbers. The have double-precision (64-bit) value. They can also represent scientific notation

```
In [53]: floatval = 9.765
In [54]: floatval1 = 7.76e-5
```

# Numeric Type

A division of integer values that does not result in a whole number, will always return a floating-point number

```
In [55]: 5/2
Out[55]: 2.5
```

The floor division operator // drops the fractional part if the result is not a whole number

```
In [56]: 5 // 2
Out[56]: 2
```

# Strings

Python has a powerful and flexible built-in string processing capabilities. You can use the string literals using single quote ‘ or double quote “

```
In [57]: a = 'We can use single quote for strings'

In [58]: b = "We can also use double quotes"
```

# Strings

For multiline strings with a line break, we can use triple quotes either ''' or """

```
In [59]: c = """
    ...: We can use Triple Quotes
    ...: for multiline string
    ...: """
```

If we count the number of lines, we will get 3 lines because the new line character `\n` is included after every line in the string

```
In [60]: c.count('\n')
Out[60]: 3
```

# Strings

In Python, the strings are immutable which means that you cannot modify a string

```
In [61]: a = 'strings are immutable'

In [62]: a[5] = 'g'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-3da0575e820d> in <module>
----> 1 a[5] = 'g'

TypeError: 'str' object does not support item assignment
```

# Strings

But you can replace a part of it and assign to another object

```
In [63]: b = a.replace('immutable','not mutable')

In [64]: b
Out[64]: 'strings are not mutable'
```

However, even after this operation, the object a is unmodified

```
In [65]: a
Out[65]: 'strings are immutable'
```

# Strings

Many Python objects can be converted into a string using the `str` function

```
In [66]: a = 4.5

In [67]: str(a)
Out[67]: '4.5'
```

# Strings

In Python, the strings are a sequence of Unicode character. Hence, we can treat them like other sequences such as lists and tuples. We shall study about them in next sessions.

```
In [68]: s = 'data science'

In [69]: list(s)
Out[69]: ['d', 'a', 't', 'a', ' ', 's', 'c', 'i', 'e', 'n', 'c', 'e']

In [70]: s[:4]
Out[70]: 'data'
```

The syntax `s[:4]` is called **slicing**. We shall study about them extensively while studying about the Python sequences.

# Strings

The **escape character** is used to specify special character such as newline `\n` or Unicode. The backslash character `\` is an escape character. If you want to write a string with backslashes, you need to escape them.

```
In [71]: s = '13\56'

In [72]: print(s)
13.

In [73]: s = '13\\56'

In [74]: print(s)
13\56
```

# Strings

If you string contains a lot of backslashes, you can preface the leading quote of the string with `r` whereas `r` stands for raw

```
In [75]: s = r'this\string\contains\many\backslashes'

In [76]: s
Out[76]: 'this\\string\\contains\\many\\backslashes'

In [77]: print(s)
this\string\contains\many\backslashes
```

# Strings

You can add two strings to concatenate them and produce a new string.

```
In [78]: a = 'This is first part '

In [79]: b = 'and this is the second part'

In [80]: a + b
Out[80]: 'This is first part and this is the second part'
```

# Strings

Strings have format method that can be used to substitute the formatted arguments into the string and produce a new string.

```
In [81]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

`{0:.2f}` means to format the first argument as a floating-point number with two decimal places.

`{1:s}` means to format the second argument as a string.

`{2:d}` means to format the third argument as an exact integer.

Substitute arguments for format parameters and pass arguments to format method:

```
In [82]: template.format(73.4560, 'Indian Rupee', 1)
Out[82]: '73.46 Indian Rupee are worth US$1'
```

# Booleans

The two Boolean values are `True` and `False`. The comparisons and other conditional expressions evaluate to `True` or `False`. We can combine the Boolean values with `and` and `or` keywords

```
In [83]: True and True
Out[83]: True

In [84]: False or True
Out[84]: True
```

# Type Casting

We can use `str`, `bool`, `int`, and `float` type that are also functions, to cast values of those types

```
In [85]: s = '3.56754'
In [86]: floatval = float(s)
In [87]: type(floatval)
Out[87]: float
In [88]: int(floatval)
Out[88]: 3
In [89]: bool(floatval)
Out[89]: True
In [90]: bool(0)
Out[90]: False
```

# None

`None` is the Python null value type. If a function does not explicitly return a value, it implicitly returns `None`:

```
In [91]: a = None

In [92]: a is None
Out[92]: True

In [93]: b = 7

In [94]: b is not None
Out[94]: True
```

# Date and Time

We can use built-in Python module datetime. It provides `datetime`, `date`, and `time` types. The `datetime` type contains the information stored in `date` and `time` types. So, `datetime` is the most commonly used type.

```
In [95]: from datetime import datetime, date, time
In [96]: dt = datetime(2021, 3, 31, 18, 55, 6)
In [97]: dt.day
Out[97]: 31
In [98]: dt.minute
Out[98]: 55
In [99]: dt.date()
Out[99]: datetime.date(2021, 3, 31)
In [100]: dt.time()
Out[100]: datetime.time(18, 55, 6)
```

# Date and Time

We can use `strftime` method to format a `datetime` as a string

```
In [101]: dt.strftime('%m/%d/%Y %H:%M')
Out[101]: '03/31/2021 18:55'
```

We can use `strptime` function to convert string into the `datetime` objects

```
In [102]: datetime.strptime('20210323', '%Y%m%d')
Out[102]: datetime.datetime(2021, 3, 23, 0, 0)
```

# Date and Time

We can calculate the difference between two `datetime` objects. The difference produces `datetime.timedelta` type.

```
In [103]: dt2 = datetime(2021, 4, 30, 21, 30)

In [104]: delta = dt2 - dt

In [105]: delta
Out[105]: datetime.timedelta(days=30, seconds=9294)

In [106]: type(delta)
Out[106]: datetime.timedelta
```

# Date and Time

The `output timedelta(days=30, seconds=9294)` indicates the offset of 30 days and 9294 seconds.

We can add a `timedelta` to a datetime to get a new `datetime`

```
In [107]: dt
Out[107]: datetime.datetime(2021, 3, 31, 18, 55, 6)

In [108]: dt + delta
Out[108]: datetime.datetime(2021, 4, 30, 21, 30)
```

# Date Time Formats

| Type | Description |
|---|---|
| %Y | Four-digit year |
| %y | Two-digit year |
| %m | Two-digit month [01, 12] |
| %d | Two-digit day [01, 31] |
| %H | Hour (24-hour clock) [00,23] |
| %I | Hour (12-hour clock) [01, 12] |
| %M | Two-digit minute [00,59] |
| %S | Second [00, 61] |
| %w | Weekday as integer [0 (Sunday), 6] |
| %U | Week number of the year [00, 53] Sunday is considered the first day of the week |
| %W | Week number of the year [00, 53] Monday is considered the first day of the week |
| %z | UTC time zone offset as +HHMM or -HHMM |
| %F | Shortcut for %Y-%m-%d (eg. 2021-3-31) |
| %D | Shortcut for %m/%d/%y (eg., 03/31/21) |

# Input

- Python 2.x has two that can be used to get input directly from a user, `input` and `raw_input`.
- Python 3.x has only `input` function.
- Each takes a string as an argument and displays it as a prompt in the shell.
- It then waits for the user to type something, followed by hitting the enter key.
- For Python3.x, the `input` function explicitly converts the input you give to type string.
- But Python 2.x `input` function takes the value and type of the input you enter as it is without modifying the type.

# Input

```
In [109]: name = input('Enter your name: ')
Enter your name: Bob Fisher

In [110]: print ('Are you really', name, '?')
Are you really Bob Fisher ?

In [111]: print('Are you really '+ name + '?')
Are you really Bob Fisher?
```

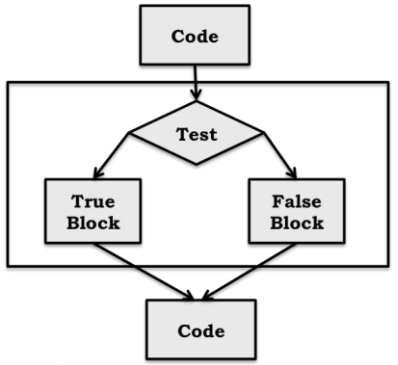
Notice that the first print statement introduces a blank before the “?” It does this because when print is given multiple arguments it places a blank space between the values associated with the arguments.

The second print statement uses concatenation to produce a string that does not contain the superfluous blank and passes this as the only argument to print.

# Control Flow

- Python has several built-in keywords for conditional logic, loops, and other standard control flow concepts found in other programming languages.

# Branching Programs




- So far, we have only looked into the **straight-line programs**. These programs execute one statement after another in the order they appear. They stop when they run out of statement.
- Another type of programs is branching programs. Conditional statement is the simplest branching statement. A conditional statement has three parts:
  - a test, i.e., an expression that evaluates to either True or False;
  - a block of code that is executed if the test evaluates to True; and
  - an optional block of code that is executed if the test evaluates to False.
- Once the conditional statement is completed, execution resumes at the code following the statement.

# Branching Programs

In Python, a conditional statement has the form

```
if Boolean expression:
        block of code
else:
        block of code
```

*Boolean expression* indicates that any expression that evaluates to `True` or `False`. It can follow the reserved word `if`.

The *block of code* indicates any sequence of Python statements that can follow `else`

# Branching Programs

Consider the following program that prints “Even” if the value of the variable x is even and “Odd” otherwise:

```
if x%2 == 0:
        print('Even')
else:
        print('Odd')
print('Done with conditional')
```

The expression `x%2 == 0` evaluates to `True` when the remainder of x divided by 2 is 0, and `False` otherwise.

Please note Remember that `==` is used for comparison, since `=` is reserved for assignment.

# Branching Programs

The conditional statements are said to be nested if either the true block or the false block of a conditional contains another conditional statement.

In the example below, there are nested conditionals in both branches of the top-level if statement.

```
if x%2 == 0:
        if x%3 == 0:
                print('Divisible by 2 and 3’)
        else:
                print('Divisible by 2 and not by 3')
elif x%3 == 0:
        print('Divisible by 3 and not by 2')
```

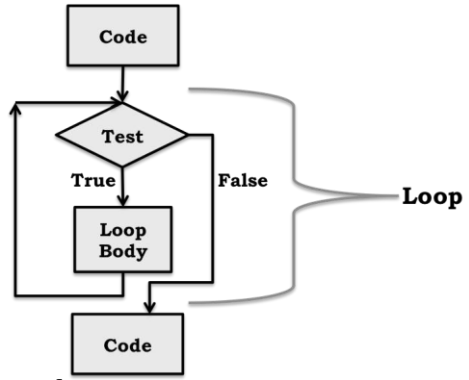
In this case, `elif` in the above code stands for “else if.”

# Branching Programs

It is often convenient to use compound Boolean expressions in the test of a conditional, for example,

```
if x < y and x < z:
        print('x is least')
elif y < z:
        print('y is least')
else:
        print('z is least')
```

# Iteration




- A generic iteration (also called looping) mechanism is depicted in Figure below.
- Like a conditional statement it begins with a test.
- If the test evaluates to True, the program executes the loop body once, and then goes back to re-evaluate the test.
- This process is repeated until the test evaluates to False, after which control passes to the code following the iteration statement.

# Iteration

Consider the following example:

```
# Square an integer, the hard way
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
        ans = ans + x
        itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

You will get answer as

```
3*3 = 9
```

# Iteration

- The code starts by binding the variable x to the integer 3. It then proceeds to square x by using repetitive addition.
- The following table shows the value associated with each variable each time the test at the start of the loop is reached.
- We constructed it by hand-simulating the code, i.e., we pretended to be a Python interpreter and executed the program using pencil and paper.
- Using pencil and paper might seem kind of quaint, but it is an excellent way to understand how a program behaves.

# Iteration

| test# | x | ans | itersLeft |
|---|---|---|---|
| 1 | 3 | 0 | 3 |
| 2 | 3 | 3 | 2 |
| 3 | 3 | 6 | 1 |
| 4 | 3 | 9 | 0 |

- The fourth time the test is reached, it evaluates to False and flow of control proceeds to the print statement following the loop.

# Iteration

- For what values of x will this program terminate?
- If `x == 0`, the initial value of `itersLeft` will also be 0, and the loop body will never be executed.
- If `x > 0`, the initial value of `itersLeft` will be greater than 0, and the loop body will be executed.
- Each time the loop body is executed, the value of `itersLeft` is decreased by exactly 1.
- This means that if `itersLeft` started out greater than 0, after some finite number of iterations of the loop, `itersLeft == 0`.
- At this point the loop test evaluates to `False`, and control proceeds to the code following the `while` statement.

# For Loops

`for` loops are for iterating over a collection (like a list or tuple) or an iterater. The standard syntax for a `for` loop is:

```
for value in collection:
# do something with value
```

You can advance a `for` loop to the next iteration, skipping the remainder of the block, using the `continue` keyword. Consider this code, which sums up integers in a list and skips `None` values:

# For Loops

`for` loops are for iterating over a collection (like a list or tuple) or an iterater. The standard syntax for a `for` loop is:

```
for value in collection:
# do something with value
```

You can advance a `for` loop to the next iteration, skipping the remainder of the block, using the `continue` keyword.

# For Loops

Consider this code, which sums up integers in a list and skips `None` values:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
        if value is None:
                continue
        total += value
```

# For Loops

A `for` loop can be exited altogether with the `break` keyword. This code sums elements of the list until a 5 is reached:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
        if value == 5:
                break
        total_until_5 += value
```

The `break` keyword only terminates the innermost for loop; any outer for loops will continue to run:

# For Loops

```
In [112]: for i in range(4):
     ...:     for j in range(4):
     ...:         if j > i:
     ...:             break
     ...:         print((i, j))
     ...:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

# While Loop

A `while` loop specifies a condition and a block of code that is to be executed until the condition evaluates to `False` or the loop is explicitly ended with `break`:

```
x = 256
total = 0
while x > 0:
        if total > 500:
                break
        total += x
        x = x // 2
```

# pass

`pass` is the “no-op” statement in Python. It can be used in blocks where no action is to be taken (or as a placeholder for code not yet implemented); it is only required because Python uses whitespace to delimit blocks:

```
if x < 0:
        print('negative!')
elif x == 0:
        # TODO: put something smart here
        pass
else:
        print('positive!')
```

# range

The `range` function returns an iterator that yields a sequence of evenly spaced integers:

```
In [113]: range(10)
Out[113]: range(0, 10)

In [114]: list(range(10))
Out[114]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# range

Both a `start`, `end`, and `step` (which may be negative) can be given:

```
In [115]: list(range(0,20,2))
Out[115]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [116]: list(range(5,0,-1))
Out[116]: [5, 4, 3, 2, 1]
```

You can use functions like `list` to store all the integers generated by range in some other data structure

# range

As you can see, `range` produces integers up to but not including the endpoint. A common use of `range` is for iterating through sequences by index:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
        val = seq[i]
```

This snippet sums all numbers from 0 to 99,999 that are multiples of 3 or 5:

```
sum = 0
for i in range(100000):
        # % is the modulo operator
        if i % 3 == 0 or i % 5 == 0:
                sum += i
```

# Ternary Expressions

A ternary expression in Python allows you to combine an `if-else` block that produces a value into a single line or expression. The syntax for this in Python is:

```
value = true-expr if condition else false-expr
```

Here, `true-expr` and `false-expr` can be any Python expressions. It has the identical effect as the more verbose:

# Ternary Expressions

```
if condition:
        value = true-expr
else:
        value = false-expr
```

This is a more concrete example:

```
In [118]: 'Non-negative' if x >= 0 else 'Negative'
Out[118]: 'Non-negative'
```

# Thanks

Samatrix Consulting Pvt Ltd

Samatrix.io