python_basic

November 16, 2020

1 Python Basic

1.1 Variables

Variables are containers for storing data values. Unlike other programming languages, a variable is created the moment you first assign a value to it. to assign a value to a variable, use the (=) sign.

Use print() to see the value assigned to the variable

[5]:
```
x = 'apple'
Y = 1
_1 = 2
print (x,Y,_1)
```

```
apple 1 2
```

Variable names in Python can be any length and can consist of uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the underscore character (_). An additional restriction is that, although a variable name can contain digits, the first character of a variable name cannot be a digit.

[6]:
```
1apple = apple
```

```
  File "<ipython-input-6-f38a6c9f9c71>", line 1
    1apple = apple
     ^
SyntaxError: invalid syntax
```

1.2 Data types

Common Data types that are used in the projects includes

Text type - String (str)

Numeric type - Integer (int) - Float (float)

Sequence Type - list - range

Using `Print(type())` we can see the data type.

1.2.1 Text type

String(str)

Strings literal are denoted by single (' ') or double quotes (" ")

Examples of String

[14]:
```
x = 'apple'
y = "pear"
print(type(x))
```

```
<class 'str'>
```

We can find the length of the variable by using the in build function len()

[15]:
```
print(len(x))
print(len(y))
```

```
5
4
```

1.2.2 Numeric type

Integer (Int)

Int are whole integers which has no decimal points and have unlimited length. Examples of Integers. Underscore are allowed between integer groupings

[36]:
```
x = 1_3
y = -2
print(type(x))
```

```
<class 'int'>
```

Wrong examples:

Example z gives a invalid token error because leading zero in a non-zero decimal number are not allowed

[32]:
```
z = 01
```

```
  File "<ipython-input-32-360f18516b30>", line 1
    z = 01
        ^
SyntaxError: invalid token
```

Float(float)

Float are Floating point numbers that contains one or more decimals

underscore are allowed between float groupings

[33]:
```
x = 1.0
y = -3.14_24
print(type(y))
```

```
<class 'float'>
```

Int can be coverted to float, vice versa.

[43]:
```
x = 1
print(type((x)))
print(x)
x = float(x)
print(type((x)))
print(x)
x= int(x)
print(type((x)))
print(x)
```

```
<class 'int'>
1
<class 'float'>
1.0
<class 'int'>
1
```

Wrong examples:

Example z gives a syntax error because as the underscore is not between digits.

[35]:
```
z = 31._3
```

```
  File "<ipython-input-35-16e45c8381ba>", line 1
    z = 31._3
           ^
SyntaxError: invalid syntax
```

1.2.3 Sequence type

List

List is a collection which is ordered and changeable.

A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index.

[11]:
```
list_example= ['a',1,'c']

print(list_example)
print(list_example[0])
```

```
['a', 1, 'c']
a
```

We can find the length of the list by using len()

[16]:
```
print(len(list_example))
```

```
3
```

1.2.4 range

The range() function is used to generate a sequence of numbers over time.

We can create a start, stop and step while using the range() function.

The syntax of range function is : range(start,stop,step)

Start is where the range start (inclusive), stop is where the range stops at (exclusive), step determines each increment

[21]:
```
print(range(5))
print(range(2,5))
print(range(1,10,2))
```

```
range(0, 5)
range(2, 5)
range(1, 10, 2)
```

1.3 Type conversion

We can convert one type of data to another with int(),float() and str().

[25]:
```
x = 1
print(type(x))

x = str(x)
print(type(x))

x = float(x)
print(type(x))

print(x)
```

```
<class 'int'>
<class 'str'>
```

```
<class 'float'>
1.0
```

However, converting data type to an invalid literal would give a invalid literal error.

[39]:
```
x = 'apple'
x = int(x)
print(type(x))
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-39-7753f23209e0> in <module>
         1 x = 'apple'
   ----> 2 x = int(x)
         3 print(type(x))


       ValueError: invalid literal for int() with base 10: 'apple'
```

1.4 Concatenation and Operations

Concatenate by using (+) operator - Multiply using (*) operator - Minus using (-) operator - Divide using (/) operator - Modulus using (%) operator - to find remainder - Exponentiation using (**) operator - Floor division using (//) operator

[17]:
```
x = 'hi'
y = 'bye'
z = 2
print(x+y)
print(x*2)
x = 3
y = 2
print(x+y)
print(x-y)
print(x*y)
print(x/y)
print(x%y)
print(x**y)
print(x//y)
```

```
hibye
hihi
5
```

```
1
6
1.5
1
9
1
```

1.4.1 Operator precedence

Operator have different precedence. It is important to take note of the precedence of the operator in order to accurately represent your equations. We can also use the Brackets to represent the precedence.

This is the operator precedence from Lowest to highest precedence.

1. +,-
2. *,/,//,%
3. **

Examples of Operator precedence

[16]:
```
print(3**2 * 3 +1)

print(2**2**3)

print((20+2*4) / 2)
```

```
28
256
14.0
```

1.4.2 Logical operators

A logical operator is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. Common logical operators include AND, OR, and NOT.

For and it

For or Returns True if one of the statements is true

For not Reverse the result, returns False if the result is true Examples:

[10]:
```
x = 5
y = 5
z = 5
print(x < 3 and x < 10)
print(y > 3 or y < 10)
print (not(z < 4))
```

```
False
True
```

```
True
```

1.5 Assignment Statement

We can (re)bind names to values and also modify attributes or mutable objects using assignment statements.

When assigning the variable x to a different value, we can see that variable x id is also changed by using the in-build function id().

However, it is possible that two objects with non-overlapping lifetimes can have the same id() value.

[21]:
```
x=1
print(id(x))
x=3
print(id(x))
print(x)
```

```
140703391719824
140703391719888
3
```

1.6 Comments

We can use write comments on our code by using the hash character(#).

A comment signifies the end of the logical line most of the time, unless implicit line joining rules are invoked.

[22]:
```
x = 1 # x is being assigned the value 1
```

[23]:
```
x = 1 \ #explicit line joining
print(x)
```

```
  File "<ipython-input-23-bda2e21dbab5>", line 1
    x = 1 \ #explicit line joining
                                  ^
SyntaxError: unexpected character after line continuation character
```

1.7 Indentation

Indentations begins at the start of the logical line which is used to determine the grouping of the statements.

Indentation are rejected if there are an inconsistent mix of tabs and spaces. This causes a Tab Error.

[26]:
```
if 1<4:
    print('yes')
else:
    print('no')
```

```
yes
```

[35]:
```
if 1>4:
    print('yes')
else:
print('no')
```

```
  File "<ipython-input-35-a7c8ff9be281>", line 4
    print('no')
    ^
IndentationError: expected an indented block
```

1.8 Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

You can pass data, known as parameters, into a function. A function can return data as a result.

We use `def` to define a function. `def` is followed by the function name and a parentheses and a colon punctuation.

This is a basic example of a function that prints a string

[ ]:
```
def my_function():
  print("Hello from a function")
```

By typing "my_function()", this would call the function and run the action it in which is to print

"Hello from a function"

[ ]:
```
my_function()
```

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

We use the return function to send back an output

Examples of Parameters: Sum(x,y)

A parameter is the variable listed inside the parentheses in the function definition.

Examples of arguments: Sum(1,2)

An argument is the value that is sent to the function when it is called.

Example of how a function with arguments and parameter.

Step 1: Create the function Step 2: Define the parameters Step 3: Action in the function Step 4: The desired output

[1]:
```
def sum(x,y):
    return (x+y)
```

Step 2: call the function and input the arguments

[2]:
```
total_sum = sum(1,2)
print(total_sum)
```

```
3
```

It is good practice to make sure you have the same number of arguments and parameter.

If we are missing an argument, there would be a TypeError stating that the function is missing a required positional argument

[5]:
```
total_sum = sum(1)
print(total_sum)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-5-303c189156ab> in <module>
----> 1 total_sum = sum(1,)
      2 print(total_sum)
      3


TypeError: sum() missing 1 required positional argument: 'y'
```

If we input more arguments than the parameter, we would also get a TypeError stating the function sum() takes 2 positional arguments but 3 were given.

[6]:
```
total_sum = sum(1,2,3)
print(total_sum)
```

```
␣
↪---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-6-5c272ec8fe3b> in <module>
----> 1 total_sum = sum(1,2,3)
      2 print(total_sum)


TypeError: sum() takes 2 positional arguments but 3 were given
```

For Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

[9]:
```
def name(*friend):
  print("The best friend is " + friend[2])

name("Bob", "Tom", "Eden")
```

```
The best friend is Eden
```

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

[11]:
```
def fruits(fruit3, fruit2, fruit1):

  print("My favourite fruit is " + fruit2)

fruits(fruit1 = "apple", fruit2 = "pear", fruit3 = "orange")
```

```
My favourite fruit is pear
```

Arbitrary Keyword Arguments

kwargs If you do not know how many keyword arguments that will be passed into your function, add two asterisk: before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

[22]:
```
%reset -f

def fruit_function(**fruit):
  print("My favourite fruit is " + fruit["fruit2"])
```

```
fruit_function(fruit1 = "apple", fruit2 = "pear")
```

```
My favourite fruit is pear
```

1.9 Loops

Loops allow the code to iterate and repeat itself within set parameters. Loops allow you to automate processes within the program using a few lines of code.

Loops typically come in two forms: 1. the For statement 2. the While statement

The for statement iterates over the elements of a sequence, such as a string, list, range, or other iterable objects. With the for loop you can automatically execute the code once for each item in the sequence.

An example of a for loop would be:

[ ]:
```
dateList = ["01-01-2000", "31-12-2020", "05-06-2018", "04-10-2010"]
for date in dateList:
    print(date)
```

The 'date' between `for` and `in` in the code is arbitary, and can be replaced by any other letter or string.

In this case "date" is known as the target_list, while the actual list of dates is known as the expression_list. The target list can also have multiple values matching the number of values within the expression_list such as:

[ ]:
```
for (i,j) in [(1,2), (3,4), (5,6)]:
    print(i)
    print(j)
```

the number of values in the target_list not matching the number of values in the expression_list will give an error:

[2]:
```
for (i,j,k) in [(1,2), (3,4), (5,6)]:
    print(i)
    print(j)
    print(k)
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-2-0e03c00705c4> in <module>
   ----> 1 for (i,j,k) in [(1,2), (3,4), (5,6)]:
```

```
2     print(i)
3     print(j)
4     print(k)
```

```
ValueError: not enough values to unpack (expected 3, got 2)
```

For loops can also be used in conjunction with the `range()` function such as in:

[ ]:
```
n = 5
for i in range(n):
    print(i)
```

On the other hand, the while statement repeatedly tests an expression and executes the code for as long as the expression is true.

An example of a while loop would be:

[ ]:
```
i = 1
while i < 6:
    print(i)
    i += 1
```

as long as i is less than 6, the code will print the current value of i before adding 1 to the value of i. When the value of i reaches 6 the loop will automatically end.

1.10 if/elif/else

`if…elif…else` statement is used for decision making in Python. If statement is represented by "if" and it will only be executed when the statement is true. Else if statement is represented by `elif` and it will only execute when the statement is true and if the previous statement was not true. Else statement is represented by `else` and it will only execute when all previous statement were not true. Note: the statements must have a boolean output and each statement blocks have to be indented.

Example:

[5]:
```
a = 200
b = 33

if b > a: #if b is greater than a, python would run the action in the if␣
 ↪condition. The following code with continue to run
  print("b is greater than a")
elif a == b: #if a and b are equal and the previous condition is false, python␣
 ↪would run the action in the elif condition.
  print("a and b are equal")
else: # python would run the else condition if all previous conditions are false
  print("a is greater than b")
```

```
a is greater than b
```

There is also an alternative way to write if/else/elif statements that are one liner.

[7]:
```
a = 200
b = 33
if b > a: print("b is greater than a")
else: print("a is greater than b")
```

```
a is greater than b
```

More examples

[11]:
```
dateIndex = 5

if dateIndex < 3: #10 is more than 3, therefore this statement is false
       print(dateIndex, 'less than 3');
elif dateIndex < 8: #since previous statement is false, it will land on elif␣
 ↪statement. Since 10 is more than 8, therefore it is false
         print(dateIndex, 'less than 8');
else: #since all previous statements is false, it will execute the else block
         print(dateIndex);
```

```
5 less than 8
```

Indentations are important in `if/else/elif` conditional statements. If the indentations are not properly used after declaring the `if/else/elif` statements, there will be an indentation error. dateIndex+1 = 5 statement wont give you an boolean output which will give you an error. The correct statement is dateIndex+1 == 5, which will give you either true or false. Bad example: If dateIndex+1 = 5: print(dateIndex); else: print(dateIndex);

1.11 Tuple

Tuple is one of the basic sequences. It is immutable and usually used to store collections of data. Tuple may be constructed in different number of ways: using a pair parentheses to denote the empty tuple: `()` Using a trailing comma for singleton tuple: 1, or `(1,)` Separating items with commas: 1, 2, 3 or `(1, 2, 3)` Using the tuple() built-in: tuple() or `tuple([1,2,3])` #tuple(iterable)

The constructor builds a tuple whose items are the same and in the same order as iterable's items. Iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a tuple, it is returned unchanged. For example, `tuple('haha')` returns `('h', 'a', 'h', 'a')` and `tuple([a, b, c])` returns `(a, b, c)`. If no argument is given, the constructor creates a new empty tuple, (). Note: Comma is what makes a tuple, not the parentheses. Parentheses are only compulsory when constructing an empty tuple or when they are needed to avoid syntactic ambiguity. Example `f(z, y, x)` #this is a function with 3 arguments `f((z, y, x))` #this is a function with a 3-tuple as the only argument

[14]:
```
tuple('haha')
```

[14]:
```
('h', 'a', 'h', 'a')
```

[16]:
```
tuple('1')
```

[16]:
```
('1',)
```

1.12 F-strings

A new string formatting mechanism known as Literal String Interpolation or more commonly as `F-strings` (because of the leading f character preceding the string literal). The idea behind `f-strings` is to make string interpolation simpler. We would input variables in {} that will be replaced with their values.

Examples:

[17]:
```
name = 'bob'
age = 23
print(f"Hello, My name is {name} and I'm {age} years old.")
```

```
Hello, My name is bob and I'm 23 years old.
```

1.13 Try Except

The try and except block in Python is used to catch and handle exceptions. The try block lets you test a block for exceptions. The except block lets you catch and handle the exceptions if there are any.

Example:

[20]:
```
num1 = 1
num2 = 0
try: #Python will try to run the code in the try block
        print(num1/num2) #There is a division by 0 situation and python raise a␣
 ↪ZeroDivisionError
except ZeroDivisionError as e:#Except block catches the ZeroDivisionError␣
 ↪exception and run the except block
        print("You cannot divide by 0" ) #Python prints "You cannot divide by 0"
```

```
You cannot divide by 0
```

1.13.1 Assert Statement

The `assert` Statement: When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception. If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError.

Examples:

We first have to `assert` a condition

[ ]:
```
assert condition
```

We then set a conditional statement to immediately trigger an error if the condition is false. The AssertionError() is a function that is predetermined by the user and it would run when the if condition is trigger.

[ ]:
```
if not condition: raise AssertionError()
```

1.14 Exception

Exceptions are errors that were detected during execution. Exception can arise even if a statement and expression is syntactically correct, the cause is due to python not being able to cope with the code. It comes in different types and the type is printed as part of the error message.

Examples:

[ ]:
```
ValueError # Raised when an operation or function receives an argument that has␣
 ↪the right type but an inappropriate value, and the situation is not␣
 ↪described by a more precise exception such as IndexError.
AssertionError #Raised when an assert statement fails

KeyError # Raised when a mapping (dictionary) key is not found in the set of␣
 ↪existing keys.

ZeroDivisionError # Raised when the second argument of a division or modulo␣
 ↪operation is zero. The associated value is a string indicating the type of␣
 ↪the operands and the operation.
```

Exception error message comes in form of a stack traceback and it shows the context where the exception occurred.

Example

[ ]:
```
#Assertion Stack traceback message
Traceback (most recent call last):
  File "/home/bafc2f900d9791144fbf59f477cd4059.py", line 4, in
    assert y!=0, "Invalid Operation" # denominator can't be 0

AssertionError: Invalid Operation
# We can tell that this is an AssertionError from the name of this exception␣
 ↪and the assertion error occurred in line 4 of the code.
```

Exceptions are not unconditionally fatal as it is possible to write programs that handle selected exceptions. To handle exceptions, Try Except is required. Look at the following example, it assigns two variables num1 and num2 with a value of 1 and 0 respectively. Python will try to execute the try block but num1 is divided by 0 which raises a ZeroDivisionError. The except block will catch the exception ZeroDivisionError and execute the except block.

[23]:
```
num1 = 1
num2 = 0
try:
```

```
        print(num1/num2)
except ZeroDivisionError as e:
        print("You cannot divide by 0")
```

```
You cannot divide by 0
```

1.15 Lambda Function

A `lambda` function is a small anonymous function. A `lambda` function can take any number of arguments, but can only have one expression.

Examples:

[24]:
```
x = lambda a : a + 10 # 5 is assigned to a, this would return a value of 15
print(x(5))
```

```
15
```

[25]:
```
x = lambda a, b, c : a + b + c # lamda can take any number of arguments.
print(x(5, 6, 2))
```

```
13
```

The power of `lambda` is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number

Examples:

[26]:
```
def myfunc(n): # we define a function with parameter (n)
  return lambda a : a * n # this returns a value of 3*11=33

mytripler = myfunc(3)

print(mytripler(11))
```

```
33
```

1.16 Classes, Object and ___ init ___

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A `Class` is like an object constructor, or a "blueprint" for creating objects. Classes allow us to bundle data and functionalities together.

To create a `class` we have to use the keyword `class`

Example:

[27]:
```
class MyClass:
  x = 5
```

Creating a new class would also create a new type of `object`. This would create a new instance of that type to be made. Now we can use the class named MyClass to create `objects`:

Examples:

[28]:
```
p1 = MyClass()
print(p1.x)
```

```
5
```

All classes have a function called '**init**()', which is always executed when the class is being initiated. Use the '**init**()' function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Create a class named Person, use the '**init**()' function to assign values for name and age

Example:

[30]:
```
class Person:
  def __init__(self, name, age): #create a function with parameters self, name,␣
↪age
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name) # this would print the
print(p1.age)
```

```
John
36
```

If one more argument is given when creating an object, there would be a "TypeError: **init**() takes 3 positional arguments but 4 were given"

Bad Example:

[31]:
```
class Person:
  def __init__(self, name, age): #create a function with parameters self, name,␣
↪age
    self.name = name
    self.age = age

p1 = Person("John", 36, 46)

print(p1.name) # this would print the
print(p1.age)
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-31-25e376e34c50> in <module>
         4     self.age = age
         5
    ----> 6 p1 = Person("John", 36, 46)
         7
         8 print(p1.name) # this would print the


       TypeError: __init__() takes 3 positional arguments but 4 were given
```

If one less argument is given when creating an object, there would be a "TypeError: **init**() missing 1 required positional argument: 'gender' "

Bad Example:

[32]:
```
class Person:
  def __init__(self, name, age, gender): #create a function with parameters␣
 ↪self, name, age
    self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name) # this would print the
print(p1.age)
```

```
         File "<ipython-input-32-e8fbb26d380d>", line 4
       self.age = age
                  ^
   TabError: inconsistent use of tabs and spaces in indentation
```

Objects can also contain methods. Methods in objects are functions that belong to the object.

Example:

[38]:
```
%reset -f

class Person:
  def __init__(self, name, age):
```

```
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36) # creates a person object with the two argument.
p1.myfunc()#call object methods myfunc(self)
```

```
Hello my name is John
```

An TypeError would occur if there is a missing argument.

Bad Example:

[39]:
```
%reset -f

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self, name):
    print("Hello input name is " + self.name)

p1 = Person("John", 36) # creates a person object with the two argument.
p1.myfunc()#call object methods myfunc(self) but there is a missing parameter
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-39-f7e5c90d73dd> in <module>
        10
        11 p1 = Person("John", 36) # creates a person object with the two␣
↪argument.
   ---> 12 p1.myfunc()#call object methods myfunc(self) but there is a missing␣
↪parameter
        13


       TypeError: myfunc() missing 1 required positional argument: 'name'
```

1.17 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. The class inheritance mechanism allows multiple base classes. A derived class can override any methods of its base class or classes. A method can call the method of a base class with the same name.

There are 2 main classes, Parent class and Child class Parent class is the class being inherited from, also called base class. Child class is the class that inherits from another class, also called derived class. We can create a parent class

Example:

[10]:
```
%reset -f
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)#Use the Person class to create an␣
 ↪object, and then execute the printname method:

x = Person("JunJie", "Edwin")
x.printname()
```

```
JunJie Edwin
```

We then create child class. This child class will inherit the properties and methods from the Person class

Example:

[11]:
```
class Student(Person):
    pass # we use pass because we don't want to add any other properties or␣
 ↪methods to class.
```

We can then create the child class (student) to create an object.

[12]:
```
x = Student("Casper", "Calvin")
x.printname() #print out the properties in the class.
```

```
Casper Calvin
```

Super() Function

Python has a function called super(). This would make the child class inherit all the methods and properties from its parent. By using super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

[15]:
```
class Student(Person):
  def __init__(self, fname, lname):
```

```
    super().__init__(fname, lname)


x = Student("Michael", "Jackson")
x.printname()
```

```
Michael Jackson
```