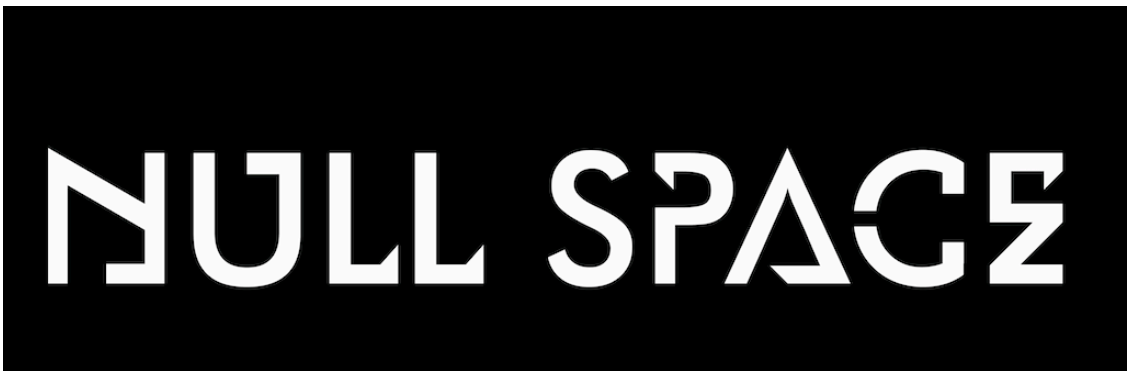
Fundamentals_Python

October 15, 2020



1 Some absolute fundamentals

A Python **program** is traditionally called a **script** and is a set of definitions and commands that are executed by the Python interpreter in something known as the **shell** - this framework converts the python commands to **machine code** that the computer hardware understands and then executes our commands. A **command** is a statement that instructs the interpreter to do something. Note that **objects** are things or data types on which we perform computations and manipulations. Some basic types are:

1. **int** - represents integer values.
2. **float** - represents real number values.
3. **bool** - represents boolean true or false notations.
4. **None** - a none type which denotes NA values.

When we combine an object and an **operator** like addition/subtraction we get an **expression**. Now note that **variables** provide us a way to associated names with objects. For example: ***pi = 4: In this the name pi binds itself to an object of int type, whose value is 4***. Further we note that the variable is just a name that can house any value.

1.1 Strings

While working with different data types we note that certain operators are said to be **overloaded**. For example if we use the + operator with numbers we get addition but if we use with strings we get concatenation.

[1]:
```
a = 4
b = 5
a+b
```

[1]: 9

[2]:
```
a = 'a'
b = 'b'
a+b
```

[2]: 'ab'

We can easily find the **length** of a string and index the string. We talk about **indexing** the string because python recognizes strings as iterable objects. Typically indexing is used to extract individual characters from a string. We also use **slicing** to extract substrings of a string by using the general form of **string[start:end]**. Examples presented below.

[3]:
```
a = 'string'
len(a)
```

[3]: 6

[4]:
```
a[2]
```

[4]: 'r'

[5]:
```
a[1:4]
```

[5]: 'tri'

1.2 Taking inputs

We can take inputs from the user by using the **input** command. Note that every value entered by the user, whether numeric or character, is automatically understood by python as a string type so we usually have to perform a **type conversion** that changes the type of string to int incase we are looking for numeric values.

[6]:
```
a = input("enter a numeric value: ")
```

```
enter a numeric value:  3
```

[8]:
```
integer_a = int(a)
integer_a
```

[8]: 3

1.3 Iteration

An iteration statement or a looping statement starts with a **test** which is essentially a boolean expression. If the test evaluates to **true** the code in the loop body is executed once, after which control goes back to the starting of the loop or the **test condition**. Again the test is evaluated and if found true, the loop is executed. This cycle keeps happening until the **test is evaluated as false**. After which, the control passes on to code statements after the loop body.

[11]:
```
# primitive way to compute square of a number by repeated addition

x = 5
square = 0
iterations = x
while (iterations!=0):
    square = square + x
    iterations = iterations - 1
print(str(x)+ "*" + str(x) + "=" + str(square))
```

```
5*5=25
```

2 Exhaustive enumeration

Now that some basics are out of the way we write some simple, fundamental programs. Find an integer cube root using exhaustive enumeration. A key point to note is that whenever we write a loop it must contain an incrementing or decrementing function since it is on this basis that the test condition is evaluated and the loop is terminated. This is called guess and check exhaustive enumeration because we tend to try each and every number (that is we exhaust all possibilities) till we get the correct answer.

[19]:
```
x = 125
ans = 0
while (ans**3 < abs(x)):
    ans = ans+1
if ans**3 != abs(x):
    print(str(x) + " is not a perfect cube")
else:
    if x < 0:
        ans = -ans
    print("the cube root is: ", ans)
```

```
the cube root is:  5
```

2.1 The for loop

When we are iterating over a sequence of integers we usually use the **for** loop. It is usually of the format **for *variable* in *sequence*:**. The iterable variable is first bound to the first value of the sequence, for which the code block is executed. The variable is then assigned the second value of the sequence and so on until the last value of the sequence. These sequences of integers that are bound to the iterable are created in python using the **range** function. It is typically of the form **range(start, stop, step)** where **step** is the value by which the sequence jumps between successive values of the sequence. Note further that a **break** statement in a loop body makes the control flow exit the inner most loop in which the break is inserted. A simple example of a nested loop is shown.

[23]:
```
x = 4
for i in range(x):
    print("Outer", "Inner")
    for j in range(x):
        print(i,"   ", j)
```

```
Outer Inner
0     0
0     1
0     2
0     3
Outer Inner
1     0
1     1
1     2
1     3
Outer Inner
2     0
2     1
2     2
2     3
Outer Inner
3     0
3     1
3     2
3     3
```

[24]:
```
# exhaustive enumeration cube root using for loop

x = 125
for ans in range(0, abs(x)+1):
    if ans**3 >= abs(x):
        break
if ans**3 != abs(x):
    print(str(x) + "is not a perfect cube")
```

```
else:
    if x < 0:
        ans = -ans
    print("the cube root is: ", ans)
```

```
the cube root is:  5
```

2.2 Approximate solutions

We will now find square roots of numbers using exhaustive enumeration using approximation. We approximate square roots because there can be cases wherein a perfect square root of a number does not exist and in that case we will approximate it by outputting a number that is **close** to the actual square root. The definition of **close enough** for a number to classify as a square root approximation is defined by the constant **epsilon**.

[26]:
```
x = 25
epsilon = 0.01
step = epsilon**2
ans = 0.0
guesses = 0

while (abs(ans**2 - x) >= epsilon) and (ans < x):
    ans += step
    guesses += 1
if (abs(ans**2 - x) >= epsilon):
    print("failed to find square root!")
else:
    print("number of guesses it took the machine: ", guesses)
    print("the square root of ", str(x), " is approximately: ", str(ans))
```

```
number of guesses it took the machine:  49990
the square root of  25  is approximately:  4.999000000001688
```

2.3 Bisection search

Now we will find the square root approximation of integer using a process called bisection search, which happens to be slightly better algorithm than the exhaustive enumeration search. Suppose that we know the square root of x to lie between 0 and some value **max**. Since the root could lie anywhere, we randomly start by checking the **middle** element in this sequence. If the answer is much larger then we know for sure that need to look to the left of this **middle** value and if the answer is much smaller then we know we need to look to the right half of the interval. In this way we keep progressing, successively checking the same condition on subintervals as they become smaller and smaller.

[29]:
```
x = 25
epsilon = 0.01
guesses = 0
low = 0
high = x
ans = (low + high)/2.0

while (abs(ans**2 - x) >= epsilon):
    print("low", str(low), "||", "high", str(high), "||", "ans", str(ans))
    guesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (low + high)/2.0

print("=========")
print("number of guesses: ", guesses)
print("finally the square root is: ", ans)
```

```
low 0 || high 25 || ans 12.5
low 0 || high 12.5 || ans 6.25
low 0 || high 6.25 || ans 3.125
low 3.125 || high 6.25 || ans 4.6875
low 4.6875 || high 6.25 || ans 5.46875
low 4.6875 || high 5.46875 || ans 5.078125
low 4.6875 || high 5.078125 || ans 4.8828125
low 4.8828125 || high 5.078125 || ans 4.98046875
low 4.98046875 || high 5.078125 || ans 5.029296875
low 4.98046875 || high 5.029296875 || ans 5.0048828125
low 4.98046875 || high 5.0048828125 || ans 4.99267578125
low 4.99267578125 || high 5.0048828125 || ans 4.998779296875
low 4.998779296875 || high 5.0048828125 || ans 5.0018310546875
=========
number of guesses:  13
finally the square root is:  5.00030517578125
```

2.4 Newton Rhapson

This is a method using which we can approximate roots of polynomial equations. We will see its demonstration in finding the real roots of a one variable polynomial. A typical polynomial of such type could be something like : $x^2 + 3x + 2$. If p denotes a polynomial expression. So when we write $p(r)$ it means that we are finding the value of the polynomial expression when $x = r$. Note that a **root** of a polynomial is a value of r such that $p(r) = 0$. Now **Newton** provided us with a convenient formula that states that if we have a random **guess** for the root of a polynomial p then the following guess would be a better approximation to the root:

$$guess_{new} = guess - \frac{p(guess)}{p'(guess)}$$

Where p' represents the first derivative of the polynomial. For example if we have a polynomial given by: $x^2 - k$. Its first derivative is: $2x$. Also suppose that our initial guess to the root of this is y. Then a better guess would be:

$$y - \frac{y^2 - k}{2y}$$

This is also known as **successive approximation** and the code is presented below.

[30]:
```
# finding x such that x**2-24 is within epsilon of 0.01

epsilon = 0.01
k = 24
guess = k/2.0

while (abs(guess**2 - k) >= epsilon):
    guess = guess - ((guess**2 - k)/(2*guess))
print("the square root approx is: ", guess)
```

```
the square root approx is:  4.8989887432139305
```

3 Functions

In Python a typical function definition is of the form: **def *function_name* (*formal parameters*)**. Consider a typical example:

[31]:
```
def maximum(x, y):
    if x>y:
        return x
    else:
        return y
maximum(4,5)
```

[31]:
```
5
```

Note here that in the above code the line **maximum(4,5)** is the **function invocation** and the numbers in the parenthesis are called **arguments** whereas the parenthesis variables in the function definition are called **formal parameters**. When a function in invoked, the arguements get attached to the formal parameters and then the function code is evaluated. Note that a function call is an **expression** and like every expression in Python it has a value, which happens to be value returned

by the function when it is invoked. Also note that the execution of the **return** statement terminates the invokation of the function. Functions provide us with two important benefits:

1. **Decomposition**: This property of functions helps us create structure, in that we can write big programs in small modules that are pretty much self contained to the specific task they are defined to perform and can also be reused in a variety of settings.
2. **Abstraction**: This is about hiding the irrelevant details. We can use functions as a black box that allow us to obtain relevant values for our computational purposes rather than worrying about how we got the value in the first place.

4 Recursion

A typical recursive definition is as follows: There is atleast one **base case** that directly specifies the result to a specific case. Then there is the **recursive case** that defines the answer in terms of some other input. Here is how **factorial computations** are examples of recursion.

1. $1! = 1$: this is the base case.
2. $(n+1)! = (n+1) * n!$: this is the recursive definition wherein the factorial of $(n+1)$ is defined in terms of the factorial of the previous number (n).

We now look at two alternatives - iterative and recursive - formulations of the factorial function. We note that the recursion terminates with the call of the base case, that is $n == 1$ in the case below. In the subsequent code block we present yet another implementation of recursion that uses the **Fibonacci sequence of numbers**. It is given by:

$$f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 3, f(4) = 5, \cdots$$

$$f(n) = f(n-1) + f(n-2), \ \forall n > 1$$

$$f(4) = f(3) + f(2) = 3 + 2 = 5$$

[34]:
```
def factI(n):
    result = 1
    while (n > 1):
        result = result * n
        n -= 1
    return result

def factR(n):
    if n==1:
        return n
    else:
        return n*factR(n-1)
```

```
print(factI(4))
print(factR(4))
```

```
24
24
```

[35]:
```
def fib(n):
    if n==0 or n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
fib(5)
```

[35]: 8

5 Tuples, lists and dicts

Tuples are ordered sequences of elements in which the component elements can be of various data types. We can perform the usual concatenate, indexing and slicing operations on them. Typically we can iterate over elements of a tuple using a for loop. In the subsequent code block the following program is implemented: find the common divisors of 20 and 100 and then sum all the divisors.

[38]:
```
t1 = (1, 2, 'three', 4)
t2 = (3, 3.5)
print(t1+t2)
print((t1+t2)[2])
print((t1+t2)[1:3])
```

```
(1, 2, 'three', 4, 3, 3.5)
three
(2, 'three')
```

[47]:
```
def find_divisors(n1, n2):
    div = () #empty tuple
    for i in range(1, min(n1, n2) + 1):
        if n1%i==0 and n2%i==0:
            div = div + (i,)
    return div

print(find_divisors(20, 100))
print("======")
print("======")

total = 0
for j in find_divisors(20, 100):
    total = total + j
```

```
print("sum of common divisors: ", total)
```

```
(1, 2, 4, 5, 10, 20)
======
======
sum of common divisors:  42
```

Lists also represent sequences of data types that can be indexed and sliced. A key difference of lists with tuples is that lists are **mutable** whereas tuples are **immutable**. Note that objects that are immutable cannot be modified whereas objects that are mutable can be modified. They are represented by square brackets.

[54]:
```
a = [1,2,"three"]
print(a)
print(a[2])
print(a[0:2])

#mutable
a = a + [1]
a.append(4)
print(a)
```

```
[1, 2, 'three']
three
[1, 2]
[1, 2, 'three', 1, 4]
```

We now note that **list comprehension** is a way to apply an operation to the values in a sequence. This method creates a new list in which each element is the result of an operation performed on the elements of a certain sequence. Examples are given below:

[55]:
```
L = [x**3 for x in range(1,8)]
L
```

[55]:
```
[1, 8, 27, 64, 125, 216, 343]
```

[57]:
```
L2 = [x**2 for x in range(1,5) if x%2 == 0]
L2
```

[57]:
```
[4, 16]
```

Lastly we have **dictionaries** that are essentially **key:value** pairs. Basically we can think of them as a list of values that can be indexed by custom defined keys. Typical examples:

[65]:
```
d1 = {"air": 1, "water": 2, "ice": 5}
d1['air'] + d1['water'] - d1['ice']
```

[65]:
```
-2
```

6 High order programming

This is a concept in which we can apply a function to each element of a list. In this approach we shall use functions as arguments to another function. Typical examples are given below. In subsequent code blocks we use the **map** functionality of python that performs the same task. It specifies as the first arugment a function and as a second argument, a collection of variables that are arguments to the first argument of map.

[58]:
```
def apply_to_each(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
    return L

L = [1,4,6,7]

print(apply_to_each(L, abs))
print(apply_to_each(L, str))
```

```
[1, 4, 6, 7]
['1', '4', '6', '7']
```

[64]:
```
a = list(map(abs, [-1,2,5,-87]))
a
```

[64]:
```
[1, 2, 5, 87]
```

7 Object Oriented Programming

First we note the definition of an **abstract data type** - it is a set of objects and operations on those objects. These objects and operations are bound together, so in essence if we pass this object in a program, we get access to not only the data attributes of the object but also to the operations that allow us to manipulate that data. We note that the specifications of these operations define an **interface** between the abstract data type and the rest of the program. This set of operations provide us with an **abstraction barrier** that essentially separates the rest of our program from the computations that might happen on the abstract data type. We implement data abstractions using **classes**. First see an example and then we will go into the details of what's what.

[66]:
```
class IntSet(object):

    def __init__(self):
        self.vals = []

    def insert(self, e):
        if e not in self.vals:
            self.vals.append(e)

    def member(self, e):
        return e in self.vals

    def remove(self, e):
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e)+" not found")

    def get_members(self):
        return self.vals[:]

    def __str__(self):
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return ('{' + result[:-1] + '}')
```

Now we note that when a function definition appears within a class definition it is called a **method**. They are usually called **method attributes** of the class. Here are the following operations that are supported in classes:

1. **Instantiation**: this is used to create instances of the class. The statement **s = IntSet()** would create a new object of type **IntSet()**. This new object is called an instance of **IntSet()**.
2. **Attribute references**: We use . to access attributes associated with the class. We say that **s.member** refers to the method **member** associated with instance **s** of type **IntSet()**.
3. We note that whenever a class is instantiated a call is automatically made to the **init** method in the class. When **s.IntSet()** is executed, the interpreter creates a new instance of type **IntSet()** and then calls **Intset().__init__** with the newly created object as the actual parameter that is bound or attached to the formal parameter **self**. Note that when this method invoked, then **IntSet.__init__** creates **vals** which is an object of type **list**, which then becomes a part of the newly created instance of type **IntSet()**. Note that this list is called a **Data attribute** of the instance of **IntSet()**.

4. Lastly we note that when we use dot notation to call a class method, such as **s.member(4)** then the object preceding the dot is implicitly passed as the first argument to the formal parameter **self** in the class definition of the member function.

Note that a class should not be confused with instances of it, just as an object of type list should not be confused with the list type itself.

[70]:
```
# implementing the classes and objects

s = IntSet()
s.insert(667)
print(s.member(667))
print(s.member(3))
print(s)
```

```
True
False
{667}
```

Some other key definitions:

1. Method attributes are typically defined in the class definition. For example **IntSet.member** is a class attribute. However when a class is instiated, **s = IntSet()** then instance attributes are created of the form **s.member**.
2. We say that when data attributes are associated with a class they are called **class variables** and when they are associated with an instance they are called **instance variables**. In the previous example **vals** is an instance variable.

8 Some popular functions

If we are iterating through a collection of objects and we want to keep track of the index as we keep iterating, then the **enumerate** function is quite handy. It returns a sequence of (i, value) tuples wherein are interested in evaluating expressions on the **value** and simultaneously keep track of the index **i**.

[1]:
```
demo = [(i, v**2) for i, v in enumerate(range(1, 8))]
demo
```

[1]:
```
[(0, 1), (1, 4), (2, 9), (3, 16), (4, 25), (5, 36), (6, 49)]
```

[3]:
```
demo2 = dict((i,v**3) for i, v in enumerate(range(1, 8)))
demo2
```

[3]:
```
{0: 1, 1: 8, 2: 27, 3: 64, 4: 125, 5: 216, 6: 343}
```

[5]:
```
l1 = ['a', 'b', 'c']
demo3 = dict((i, v) for i, v in enumerate(l1))
demo3
```

[5]:
```
{0: 'a', 1: 'b', 2: 'c'}
```

The **Zip** function pairs up values in lists, tuples and other types of sequences and outputs a list of tuples.

[7]:
```
a1 = ['ham', 'man', 'train']
a2 = ['45', '56', '1234']
list(zip(a1, a2))
```

[7]:
```
[('ham', '45'), ('man', '56'), ('train', '1234')]
```

[8]:
```
l1 = ['apple', 'mango', 'banana']
l2 = [3, 4, 5]
mapping = {}
for key, value in zip(l1, l2):
    mapping[key] = value
mapping
```

[8]:
```
{'apple': 3, 'mango': 4, 'banana': 5}
```

The **Lambda** function format is a way of writing short, simple functions in one simple line of code.

[9]:
```
def apply_to_list(a_list, f):
    return [f(x) for x in a_list]

mylist = [1,2,3,4,5]
apply_to_list(mylist, lambda x: x**2+(4*x)-2)
```

[9]:
```
[3, 10, 19, 30, 43]
```