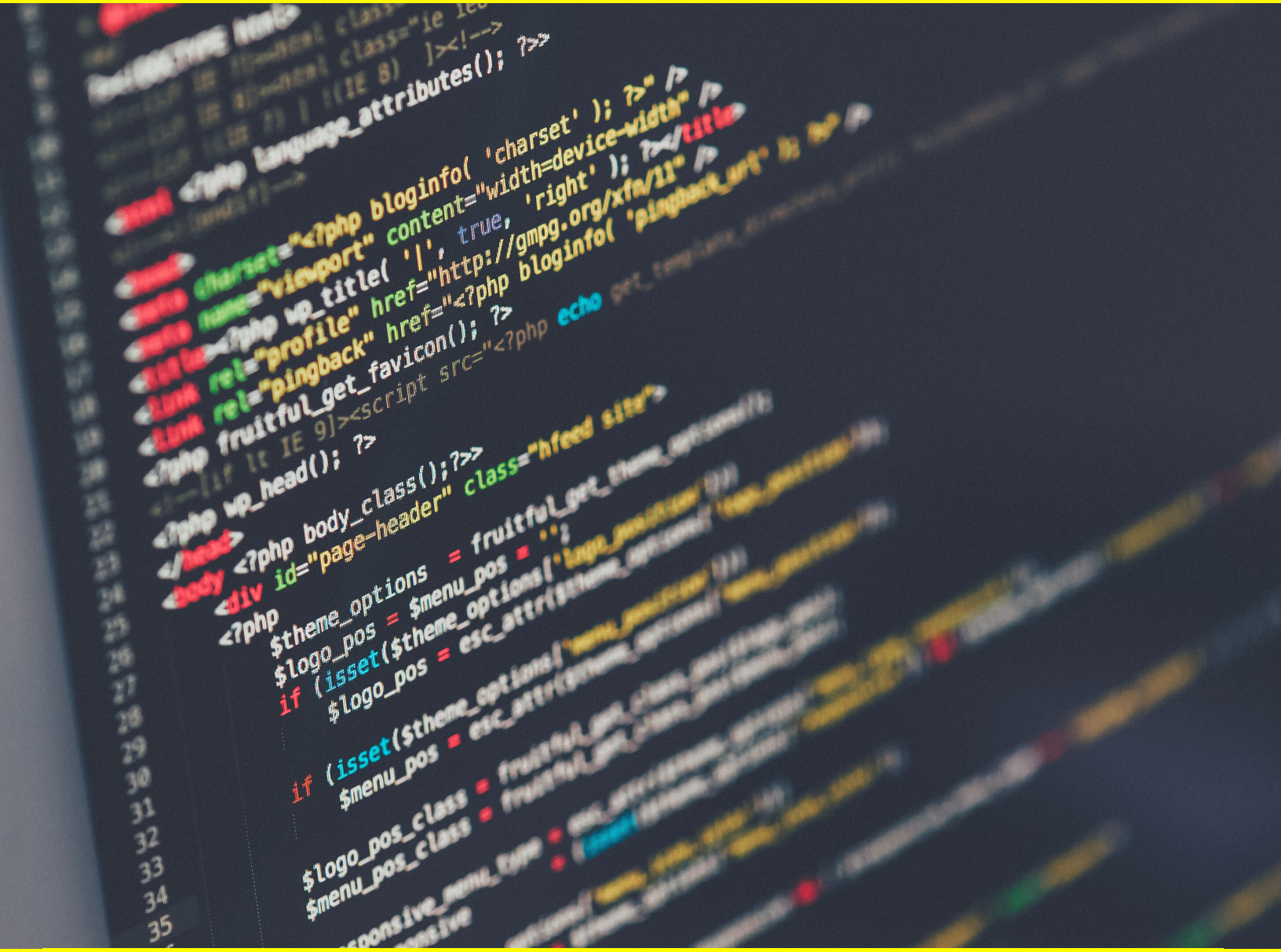
the fast lane to python

PART 6



BY:

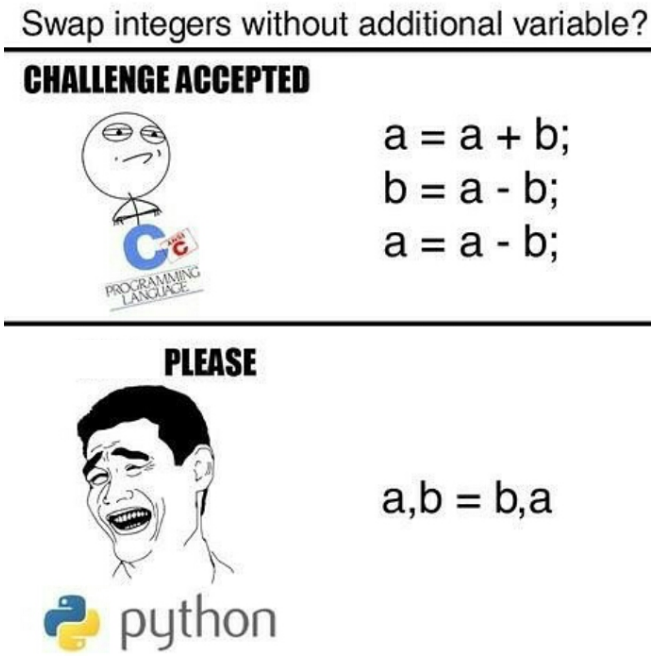
Arshadul Shaikh

Ref taken from Coursera + Udemy

STAY DEDICATED, IT WON'T HAPPEN OVERNIGHT !!!

Which is what we all have realized by now and we are here on part 6 to make a change

Appreciation is what you deserve, but from now on we will be dealing with some core topics so need your 200% in everything.



Let's roll

Dictionary

People who have learnt C++ would have heard + used Maps in their coding, somewhat similar yet advanced version of maps is Dictionary. It is an **unordered collection of data values, used to store data values** (focus on the term unordered). Unlike maps where the keys have to be the same type, and all of the values have to be the same type., Dictionary holds **key:value pair** , where key and value pair could be of any type.

Let's check one C++ program of map:

```
#include <iostream>
#include <map>
#include <string>
int main()
{
   //create a map that stores strings indexed by strings
   std::map<std::string, std::string> m;
   //add some items to the map
   m["cat"] = "mieow";
   m["dog"] = "woof";
   m["horse"] = "neigh";
   m["fish"] = "bubble";
   //now loop through all of the key-value pairs
   //in the map and print them out
   for ( auto item : m )
   {
      //item.first is the key
      std::cout << item.first << " goes ";
      //item.second is the value
      std::cout << item.second << std::endl;
   }
```

```
//finally, look up the sound of a cat
std::cout << "What is the sound of a cat? " << m["cat"]
<< std::endl;
return 0;
}
```

Output

```
cat goes mieow
dog goes woof
fish goes bubble
horse goes neigh
What is the sound of a cat? mieow
```

What you can understand from the above program?

- You need to include the <map> header file as std::map<> is part of the C++ standard library.
- You must specify the type of the key and the type of the value, between the angle brackets while defining the map
  e.g. std::map<int,double> would be a map that uses integer keys to look up double values.
- There is no .insert() function. You can only add items using lookup.
- We saw all keys in a map had the same type, and all values too had the same type, buy keys and values were of different type.
  For example, std::map<float,std::string> creates a map in which all keys are floats, and all values are strings.

Now let us try to get the same results using python dictionary:

```python
m={}
m["cat"]="mieow"
m["dog"]="woof"
m["horse"] = "neigh";
m["fish"] = "bubble";
for i in m:
    print(i + ' goes ' + m[i])
print('what is the sound of the cat?',m["cat"])
```

```
cat goes mieow
dog goes woof
horse goes neigh
fish goes bubble
what is the sound of the cat? mieow
```

definitely looks clean & easy to understand than c++
Let's jump in and try to understand Dictionaries in a better way

Getting Started with Dictionaries

One way to create a dictionary is to start with an empty dictionary and add key-value pairs. Which is what we did in above example.

- The empty dictionary is denoted {}.

- The first assignment creates an empty dictionary named **m**.
- The other assignments adds first new key-value pairs to the dictionary. The left hand side gives the dictionary and the key being associated.
  like m["cat"]
- The right hand side gives the value being associated with that key.
  like m["cat"] = "mieow"

The key-value pairs of the dictionary are separated by commas as you can see in above image. Each pair contains a key and a value **separated by a colon.**
The order of the pairs always may not be what you expect. Python uses complex algorithms, designed for very fast access, to determine where the key-value pairs are stored in a dictionary. So as in above example though we see the prints after for loop was run over dictionary m, which is our iterable, happened sequentially as we inserted but remember Dictionary unlike Lists/tuples/strings are not dependent on index algorithm but works on key value pairs.

So try to consider the output ordering of dictionary as something which is unpredictable but printed in the best possible way .

Another way to create a dictionary is to provide a bunch of key-value pairs using the same syntax as the previous output.

```
1 m={'cat': 'mieow', 'dog': 'woof', 'horse': 'neigh', 'fish': 'bubble'}
2 print(m)
```

```
{'cat': 'mieow', 'dog': 'woof', 'horse': 'neigh', 'fish': 'bubble'}
```

Dictionary Operations

del operation:
del statement removes a key-value pair from a dictionary

```
m={'cat': 'meow', 'dog': 'woof', 'horse': 'neigh' ,'fish': 'bubble'}
print(m)
del m['horse']
print(m)
```

```
{'cat': 'meow', 'dog': 'woof', 'horse': 'neigh', 'fish': 'bubble'}
{'cat': 'meow', 'dog': 'woof', 'fish': 'bubble'}
```

This operation shows what????

It shows that dictionaries are mutable.
As we've seen before with lists, this means that the dictionary can be **modified by referencing an association** on the left hand side of the assignment statement.

If it is mutable that would mean unlike tuple, we can change the values too?

```
m={'cat': 'meow', 'dog': 'woof', 'horse': 'neigh' ,'fish': 'bubble'}
print(m)
m['horse']='I dont belong here'
print(m)
```

```
{'cat': 'meow', 'dog': 'woof', 'horse': 'neigh', 'fish': 'bubble'}
{'cat': 'meow', 'dog': 'woof', 'horse': 'I dont belong here', 'fish': 'bubble'}
```

```
m={'cat': 'meow', 'dog': 'woof', 'horse': 'neigh' ,'fish': 'bubble'}
print(m)
m['horse']=m['horse'] + " " + ',i am not a pet'
print(m)
```

```
{'cat': 'meow', 'dog': 'woof', 'horse': 'neigh', 'fish': 'bubble'}
{'cat': 'meow', 'dog': 'woof', 'horse': 'neigh ,i am not a pet', 'fish': 'bubble'}
```

```
mydict = {"cat":12, "dog":6, "elephant":23}
mydict["mouse"] = mydict["cat"] + mydict["dog"]
print(mydict["mouse"])
```

```
18
```

the above program should have given error if we were trying it on any other language as mouse key does not exist at all in the dictionary. But this is python, and it understands we forget things so it added the value for cat and the value for dog (12 + 6) and created a new entry for mouse.

Dictionary Methods

This is an exciting section and you would need to practice more than once to remember all these methods.

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

- The empty parentheses in the case of keys indicate that this method takes no parameters.
    - If x is a variable whose value is a dictionary, x.keys is the method object, and x.keys() invokes the method, returning a view of the value

```
#remember dictionary did  not output the values in sequence before 3.7
#you might see random order with big dictionaries
#but it will print all key-item pairs

#post 3.7 it will always come in sequence and from 3.8 you can even
#reverse a dictionary

zoo = {"lion":12, "tiger":6, "elephant":23}

for anim in zoo: #Dictionary variable is considered same as
                 # dictionary.keys()  method
    print(anim)
    print("there are " + str(zoo[anim]) + " " + anim + " in the zoo")


for anim in zoo.keys(): # this method does not take any parameter
    print(anim)
    print("there are " + str(zoo[anim]) + " " + anim + " in the zoo")


#in above program zoo is the dictionary
# zoo.keys is the method object
#zoo.keys() is the way we will invoke the method.
#always remember () invokes the method

zoo_list= zoo.keys()
print(zoo_list)
```

```
lion
there are 12 lion in the zoo
tiger
there are 6 tiger in the zoo
elephant
there are 23 elephant in the zoo
lion
there are 12 lion in the zoo
tiger
there are 6 tiger in the zoo
elephant
there are 23 elephant in the zoo
```

```python
zoo = {"lion":12, "tiger":6, "elephant":23}

zoo_list= zoo.keys() #here the zoo_list directly takes

zoo_list=zoo.keys() #this will call dist_key object
#the dict.keys() method returns a dictionary view object,
#which acts as a set. Iterating over the dictionary
#directly also yields keys, so turning a dictionary into
#a list results in a list of all
#we will learn about sets in later part

print(zoo_list)

zoo_list=list(zoo.keys())
print(zoo_list)
```

```
dict_keys(['lion', 'tiger', 'elephant'])
['lion', 'tiger', 'elephant']
```

As we saw in earlier example that it's so common to iterate over the keys in a dictionary that you can **omit the keys method call** in the for loop — iterating over a dictionary implicitly iterates over its keys.

The **values** and **items methods** are **similar to keys**.
They return the **objects** which can be **iterated over**. Note that the item objects are tuples containing the key and the associated value.

***KEY POINT

- Technically, .keys(), .values(), and .items() don't return actual lists. Like the range function we saw earliers, in python 3 they return objects that produce the items one at a time, rather than producing and storing all of them in advance as a list.
- Unless the dictionary has a whole lot of keys, this won't make a difference for performance. In any case, as with the range function, it is safe for you to think of them as returning lists, for most purposes.
- if you **print out type(inventory.keys()),** you will find that it is something other than an actual list but dist_keys/values/items.
- If you want to get the first key, inventory.keys()[0] will fail, cause dist_key class does not support indexing, you need to make the collection of keys into a real list before using [0] to index into it: **list(inventory.keys())[0].**

```
1  inventory = {'mouse': 430, 'desktop': 312, 'keyboard': 525,
2               'headphones': 217}
3
4  print(type(inventory.keys()))
5  print(type(inventory.items()))
6  print(type(inventory.values()))
7  lit=['12']
8  print(type(lit))
9
10
11 print(list(inventory.keys())[0])
12
13 print((inventory.keys())[0])
14
```

```
<class 'dict_keys'>
<class 'dict_items'>
<class 'dict_values'>
<class 'list'>
mouse

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-bdfb15fb0ad7> in <module>
     11 print(list(inventory.keys())[0])
     12
---> 13 print((inventory.keys())[0])

TypeError: 'dict_keys' object is not subscriptable
```

let's try to use:

- in
- not in

operators.

```
inventory = {'mouse': 430, 'desktop': 312, 'keyboard': 525,
             'headphones': 217}

print('desktop' in inventory)
print('laptop' in inventory)

if 'USB' in inventory:
    print('USB is available')
else:
    print('Sorry please check in another store')
```

```
True
False
Sorry please check in another store
```

But there is a possibility that you might end up in a runtime error in older versions of python. TO fix this we have another method **get** which allows you to access the value associated with a key similar to []operator.

The important difference is that get will never cause a runtime error even if the key value is not present. Rather it will return none.

```
inventory = {'mouse': 430, 'desktop': 312, 'keyboard': 525,
             'headphones': 217}

print(inventory.get('mouse'))
print(inventory.get('laptop'))
print(inventory.get('USB',0))

```

```
430
None
0
```

```
total = 0
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
for akey in mydict:
    print(akey)
    if len(akey) > 3:
        print(len(akey))
        total = total + mydict[akey]
print(total)
```

```
cat
dog
elephant
8
bear
4
43
```

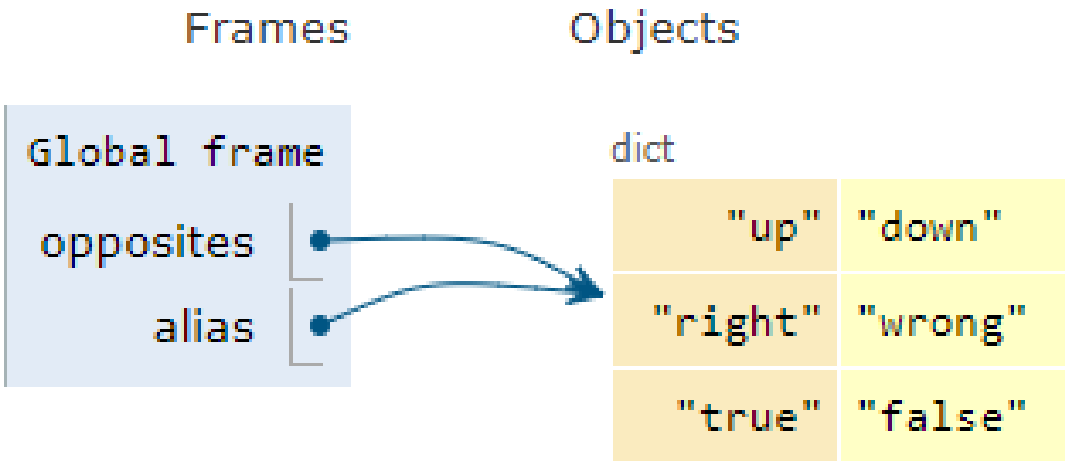
In above example, the for statement iterates over the keys. It adds the values of the keys that have length greater than 3.

Aliasing and Copying

Because dictionaries are mutable, you need to be aware of aliasing (as we saw with lists). Whenever two variables refer to the same dictionary object, changes to one affect the other. For example, opposites is a dictionary that contains pairs of opposites.

```
opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
alias = opposites

print(alias is opposites)

alias['right'] = 'left'
print(opposites['right'])
```

```
True
left
```



As you can see from the is operator, alias and opposites refer to the same object.

If you want to modify a dictionary and keep a copy of the original, **use the dictionary copy method.**
Since a copy is acopy of the dictionary, changes to it will not effect the original.

```
 8 acopy = opposites.copy()
 9 acopy['right'] = 'down'     # does not change opposites
10 print(opposites['right'])
11 print(acopy['right'])
```

```
left
down
```

Accumulation: Accumulating Multiple Results in a Dictionary

We have previously see the accumulator pattern in depth while learning

list. **Accumulator goes through the items in a sequence, updating an accumulator variable each time**. Rather than accumulating a single result, it's possible to accumulate many results.

Lets try to use a file that we used earlier while learning file handling and count number of a letter.

```
file_name=open("C:\Japan_olympics_2020.txt",'r')

# test is one long string containing all the characters
test=file_name.read()

t_count=0 #initialize the accumulator variable
p_count=0
for c in test:
    if c == 't':
        t_count = t_count + 1 #increment the counter
    if c =='p':
        p_count = p_count + 1
print("t: " + str(t_count) + " occurences")
print("p: " + str(p_count) + " occurences")

file_name.close()


```

```
t: 87 occurences
p: 6 occurences
```

Above program is good and all , but you can see this is going to get tedious if we try to accumulate counts for all the letters.

- We will have to initialize a lot of accumulators and there will be a very long if..elif..elif statement.
- Using a dictionary, we can do a lot better.One dictionary can hold all of the accumulator variables.
- Each key in the dictionary will be one letter, and the corresponding value will be the count so far of how many times that letter has occurred.

Lets' see an example

```
file_name=open("C:\Japan_olympics_2020.txt",'r')

# test is one long string containing all the characters
test=file_name.read()
x={}
x['t']= 0 #this key will use as an accumulator later
x['p']= 0 #this key will use as an accumulator later
for c in test:
    if c == 't':
        x['t'] = x['t'] + 1 #increment the counter
    if c =='p':
        x['p'] = x['p'] + 1
print("t: " + str(x['t']) + " occurences")
print("p: " + str(x['p']) + " occurences")

file_name.close()


```

```
t: 87 occurences
p: 6 occurences
```

This hasn't really improved things yet, Whichever character we're seeing, t or s, we're incrementing the counter for that character.

We need to be more efficient in writing the programs to make it more generic, let's try to see different apporach.

```
file_name=open("C:\Japan_olympics_2020.txt",'r')

# test is one long string containing all the characters
test=file_name.read()
x={}
x['t']= 0 #this key will use as an accumulator later
x['p']= 0 #this key will use as an accumulator later
for c in test:
    if c == 't':
        x[c] = x[c] + 1 #increment the counter
    if c =='p':
        x[c] = x[c] + 1
print("t: " + str(x['t']) + " occurences")
print("p: " + str(x['p']) + " occurences")

file_name.close()


```

```
t: 87 occurences
p: 6 occurences
```

You understood the change?

If yes, you are master :D , if not read on

Let's try to understand the program step by step:

- Firstly as with all assignment statements, the right side is evaluated first. In this case x[c] has to be evaluated. As with all expressions, we first have to substitute values for variable names.
- x is a variable bound to a dictionary. c is a variable bound to one letter from the string that test is bound to (that's what the for statement says to do: execute lines 9-12 once for each character in test , with the variable c bound to the current character on each iteration.)
- So, let's suppose that the current character is the letter p (we are on line 12). Then x[c] looks up the value associated with the key 'p' in the dictionary x. If all is working correctly, that value should be the number of times 'p' has previously occurred.
- For the sake of argument, suppose it's 5. Then the right side evaluates to 5 + 1, 6
- Now we have assigned the value 6 to x[c]. That is, in dictionary x, we set the value associated with the key 'p' (the current value of the variable c) to be 6. In other words, we have incremented the value associated with the key 's' from 5 to 6.

We can do better still. One other nice thing about using a dictionary is that we don't have to prespecify what all the letters will be.

In this case, we know in advance what the alphabet for English is, but later in the chapter we will count the occurrences of words, and we do not know in advance all the of the words that may be used. Rather than pre-specifying which letters to keep accumulator counts for, we can start with an empty dictionary and add a counter to the dictionary each time we encounter a new thing that we want to start keeping count of.

```
1  file_name=open("C:\Japan_olympics_2020.txt",'r')
2
3  # test is one long string containing all the characters
4  test=file_name.read()
5
6  #we are starting with an empty dictionary
7  x={}
8  for c in test:
9      if c not in x:
10
11         # we have not seen this character before,
12         #so initialize a counter for it
13         x[c] = 0
14
15     #whether we've seen it before or not, increment its counter
16     x[c] = x[c] + 1
17
18 print("t: " + str(x['t']) + " occurences")
19 print("p: " + str(x['p']) + " occurences")
20 print("s: " + str(x['s']) + " occurences")
21 print("a: " + str(x['a']) + " occurences")
22
23 file_name.close()
24
25
```

```
t: 87 occurences
p: 6 occurences
s: 54 occurences
a: 82 occurences
```

Notice that in the for loop, we no longer need to explicitly ask whether the current letter is an ‘p’ or ‘t’. The increment step on line 16 works for the counter associated with whatever the current character is. Our code is now accumulating counts for all letters, not just ‘p’ and ‘t’.

Note that the print statements at the end pick out the specific keys ‘t’ and ‘p’. We can generalize that, too, to print out the occurrence counts for all of the characters, using a for loop to iterate through the keys in x.

```
file_name=open("C:\Japan_olympics_2020.txt",'r')

# test is one long string containing all the characters
test=file_name.read()

#we are starting with an empty dictionary
x={}
for c in test:
    if c not in x:

        # we have not seen this character before,
        #so initialize a counter for it
        x[c] = 0

    #whether we've seen it before or not, increment its counter
    x[c] = x[c] + 1

for c in x:
    print(c + ':' + str(x[c]) + " count")

file_name.close()
```

```
N:30 count
a:82 count
m:5 count
e:84 count
,:120 count
S:40 count
x:1 count
```

Question 1:
Provided is a string saved to the variable name sentence. Split the string into a list of words, then create a dictionary that contains each word and the number of times it occurs. Save this dictionary to the variable name word_counts.

sentence = "The dog chased the rabbit into the forest but the rabbit was too quick."

Question 2:
Create a dictionary called char_d from the string stri, so that the key is a character and the value is how many times it occurs.

stri = "Hey, I am becoming really good in python"

```
sentence=sentence.split()
word_counts={}
for i in sentence:
    if i not in word_counts:
        word_counts[i]=0
    word_counts[i]=word_counts[i]+1
print(word_counts)


char_d={}
for i in stri:
    if i not in char_d:
        char_d[i]=0

    char_d[i]=char_d[i] +1
print(char_d
```

Accumulating results from a Dictionary + Implementation

we can also iterate through the keys in a dictionary, accumulating a result that may depend on the values associated with each of the keys.

```
sentence = "the dog chased the rabbit into the forest but the rabbit was too quick"

print(sentence)
word_counts={}

for i in sentence:
    if i not in word_counts:
        word_counts[i]=0
    word_counts[i]=word_counts[i]+1

print(word_counts)
del word_counts[' ']

letter_values = {'a': 1, 'b': 3, 'c': 3, 'd': 2, 'e': 1, 'f':4, 'g': 2, 'h':4, 'i':1,
                 'j':8, 'k':5, 'l':1, 'm':3, 'n':1, 'o':1, 'p':3, 'q':10, 'r':1, 's':1,
                 't':1, 'u':1, 'v':4, 'w':4, 'x':8, 'y':4, 'z':10}

tot=0
for i in word_counts:
    if i in word_counts:
        tot = tot + word_counts[i] * letter_values[i]
print(tot)
```

```
the dog chased the rabbit into the forest but the rabbit was too quick
{'t': 10, 'h': 5, 'e': 6, ' ': 13, 'd': 2, 'o': 5, 'g': 1, 'c': 2, 'a': 4, 's': 3, 'r': 3, 'b': 5,
'i': 4, 'n': 1, 'f': 1, 'u': 2, 'w': 1, 'q': 1, 'k': 1}
108
```

- In above example, we are trying to calculate the value as scrabble score. each occurrence of the letter ‘e’ earns one point, but ‘q’ earns 10.

- We have a second dictionary, stored in the variable letter_values.

- Now, to compute the total score, we start an accumulator at 0 and go through each of the letters in the counts dictionary. For each of those letters that has a letter value (no points for spaces, punctuation, capital letters, etc.), we add to the total score.

Accumulating the Best Key

WE have always loved to find the maximum and minimum values, most of the time in our mind itself to challenge our brain, here as well we will try to challenge our brain and try to find key associated with the maximum and minimum value.

It would be nice to just find the maximum value as we do with key, and then look up the key associated with it, but dictionaries don't work that way. **You can look up the value associated with a key, but not the key associated with a value**. (The reason for that is there may be more than one key that has the same value).

what is the trick here?

The trick is to have the accumulator keep track of the best key so far instead of the best value so far.

- let's assume that there are at least two keys in the dictionary. Then, similar to our first version of computing the max of a list, we can initialize the best-key-so-far to be the first key, and loop through the keys, replacing the best-so-far whenever we find a better one.

Write a program that finds the key in a dictionary that has the maximum value. If two keys have the same maximum value, it's OK to print out either one.

```
d = {'a': 194, 'b': 54, 'c':34, 'd': 44, 'e': 312, 'full':31}

ks = (d.keys())
# initialize variable best_key_so_far to be the first key in d
max_value=d['a']
for k in ks:
    if d[k] > max_value:
        max_value= d[k]
print("key " + best_key_so_far + " has the highest value, " + str(d[best_key_so_far]))

ks = d.keys()
best_key_so_far = list(ks)[0]  # Have to turn ks into a real list
                               #before using [] to select an item
for k in ks:
# check if the value associated with the current key is
# bigger than the value associated with the best_key_so_far
# if so, save the current key as the best so far

    if d[k] > d[best_key_so_far]:
        best_key_so_far = k

print("key " + best_key_so_far + " has the highest value, " + str(d[best_key_so_far]))

```

```
key e has the highest value, 312
key e has the highest value, 312
```

as we see we have solved the question in 2 ways

First way: we did not convert the dict_key into a list we just used it as an iterator over keys in for loop and manually assigned the first value of the dictionary to be max_Value which is correct here but not when we are not aware how will the dictionary look like while writing big programs

Second way:
we have converted the value into list and then used index to assign the max value

Question 1:
Create a dictionary called d that keeps track of all the characters in the string placement and notes how many times each character was seen. Then, find the key with the lowest value in this dictionary and assign that key to min_value.

placement = "Beaches are cool places to visit in spring however the Mackinaw Bridge is near. Most people visit Mackinaw later since the island is a cool place to explore."

Question 2:
Create a dictionary called lett_d that keeps track of all of the characters in the string product and notes how many times each character was seen. Then, find the key with the highest value in this dictionary and assign that key to max_value.

product = "iphone and android phones"

Solution:

```
d={}

for i in placement:
   if i not in d:
      d[i]=0
   d[i]= d[i] + 1
print(d)

ks=d.keys()
min_value=list(d.keys())[0]
for i in ks:
   if d[i] < d[min_value]:
      min_value=d[i]
```

```
lett_d={}

for i in product:
    if i not in lett_d:
        lett_d[i]=0
    lett_d[i]= lett_d[i] + 1

max_value=list(lett_d.keys())[0]

for i in lett_d:
    if lett_d[i] > lett_d[max_value]:
        max_value=i
print(max_value)
```

BIG QUESTION
When to use a dictionary?

- **W**hen a piece of data consists of a set of properties of a single item, a dictionary is often better. You could try to keep track mentally that the zip code property is at index 2 in a list, but your code will be easier to read and you will make fewer mistakes if you can look up mydiction['zipcode'] than if you look up mylst[2].
- When you have a collection of data pairs, and you will often have to look up one of the pairs based on its first value, it is better to use a dictionary than a list of (key, value) tuples. With a dictionary, you can find the value for any (key, value) tuple by looking up the key. With a list of tuples you would need to iterate through the list, examining each pair to see if it had the key that you want.
- On the other hand, if you will have a collection of data pairs where multiple pairs share the same first data element, then you can't use a dictionary, because a dictionary requires all the keys to be distinct from each other.

```
lett_d={}

for i in product:
    if i not in lett_d:
        lett_d[i]=0
    lett_d[i]= lett_d[i] + 1

max_value=list(lett_d.keys())[0]

for i in lett_d:
    if lett_d[i] > lett_d[max_value]:
        max_value=i
print(max_value)
```

BIG QUESTION
When to use a dictionary?

- **W**hen a piece of data consists of a set of properties of a single item, a dictionary is often better. You could try to keep track mentally that the zip code property is at index 2 in a list, but your code will be easier to read and you will make fewer mistakes if you can look up mydiction['zipcode'] than if you look up mylst[2].
- When you have a collection of data pairs, and you will often have to look up one of the pairs based on its first value, it is better to use a dictionary than a list of (key, value) tuples. With a dictionary, you can find the value for any (key, value) tuple by looking up the key. With a list of tuples you would need to iterate through the list, examining each pair to see if it had the key that you want.
- On the other hand, if you will have a collection of data pairs where multiple pairs share the same first data element, then you can't use a dictionary, because a dictionary requires all the keys to be distinct from each other.

dictionary

A collection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.keyA data item that is mapped to a value in a dictionary.

Keys

are used to look up values in a dictionary.

value

The value that is associated with each key in a dictionary.

key-value pair

One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

mapping type

A mapping type is a data type comprised of a collection of keys and associated values. Python's only built-in mapping type is the dictionary. Dictionaries implement the associative array abstract data type.

Functions

Yes we have finally reached the part where real programming would start and we are going to start using our efficient programming skills for solving big problems.

In Python, a function is a chunk of code that performs some operation that is meaningful for a person to think about as a whole unit, for example calculating a student's GPA in a learning system or responding to the jump action in a video game. Once a function has been defined and you are satisfied that it does what it is supposed to do, you will start thinking about it in terms of the larger operation that it performs rather than the specific lines of code that make it work.

Let's try to understand this in real life example:

Breaking down of a task or problem is crucial for successful implementation of any program which has more than 50 or so lines. Like, the program that displays the Instagram landing page is made up of functions that:

- display the header bar
- display your friends' posts
- display your friends' stories
- display the ad at the bottom of the screen recommending you use the app

And each of those is made up of functions as well. For example, the function that displays your friends' posts is a for loop that calls a function to:

- display a single post which in turn calls functions to:
  - display the photo and name of the person posting the story
  - display the photo itself
  - display other users' "likes" to the story
  - display the comments on the story, etc

Function Definition

The syntax for creating a named function, a function definition, is:

```
def name( parameters ):
   statements
```

- **def** is the keyword which tells that it is a function

- **name -** We can make up any names we want for the functions we create, except that we **can't use** a **name** that is a **Python keyword**, and the names must follow the rules for legal identifiers that we have already went through in PART 1.

- The **parameters**(could be empty) **specify what information**,if any, you have **to provide** in order to use the new function. Another way to say this is that the parameters specify what the function needs to do its work.

- There can be any **number of statements** inside the function, but they have to be **indented from the def**.

Standard format goes like:

- A header line which begins with a keyword def and ends with a colon.
- A body consisting of one or more Python statements, each indented the same amount – 4 spaces is the Python standard – from the header line.

```
def hello():
    """This function says hello and greets you"""
    print("Hello")
    print("Glad to meet you")
hello()
```

```
Hello
Glad to meet you
```

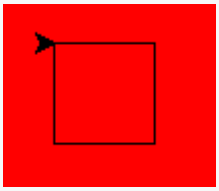
docstrings

If the first thing after the function header is a string (some tools insist that it must be a triple-quoted string), it is called a docstring and gets special treatment in Python and in some of the programming tools.
Another way to retrieve this information is to use the interactive interpreter, and enter the expression <function_name>.__doc__, which will retrieve the docstring for the function. So the string you write as documentation at the start of a function is retrievable by python tools at runtime. **This is different from comments in your code**, which are completely eliminated when the program is parsed.By convention, Python programmers use docstrings for the key documentation of their functions.

Let's try to implement our function knowledge with turtle (you would need to recall concepts of turtle, so try to give a short look over part 2

Write a program to draw a sqaure using function:

```
1  import turtle
2
3  def drawSquare(t,side):
4      """docstring for drawing a Square
5      it will help the passed turtle to
6      draw a square"""
7
8      for i in range(4):
9          t.forward(side)
10         t.right(90)
11
12
13 wn=turtle.Screen()
14 wn.bgcolor("red")
15
16 turt=turtle.Turtle()
17 drawSquare(turt,50)
18
19 wn.exitonclick()
```



How did we achieve the square?

- we defined a function named drawSquare.
- It has two parameters —
  - one to tell the function which turtle to move around
  - other to tell it the size of the square we want drawn.
  - In the function definition they are called t and side respectively.

Make sure you know where the body of the function ends — it depends on the indentation and the blank lines don't count for this purpose!

Also we are v well aware that **defining** a new function does **not make the function run**. To **execute the function**, we need a **function call**. This is also known as a **function invocation**.

The way to invoke a function is to refer to it by name, followed by parentheses

How did we achieve the square?

- we defined a function named drawSquare.
- It has two parameters —
  - one to tell the function which turtle to move around
  - other to tell it the size of the square we want drawn.
  - In the function definition they are called t and side respectively.

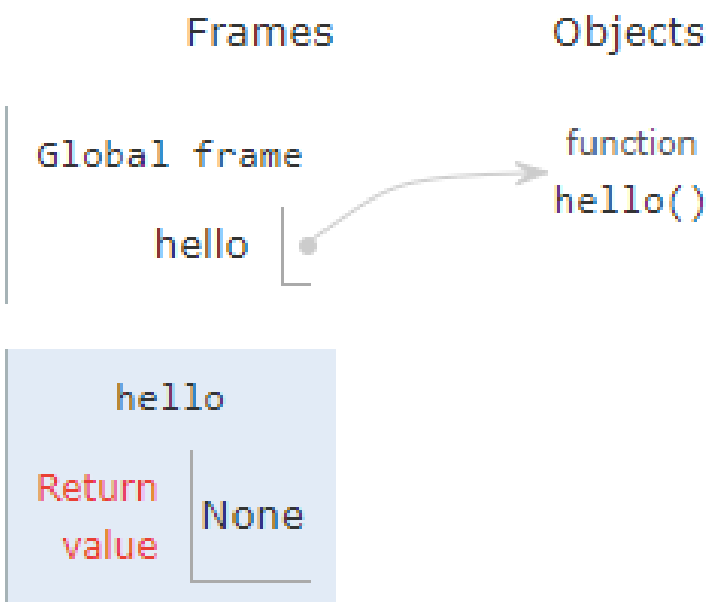
Make sure you know where the body of the function ends — it depends on the indentation and the blank lines don't count for this purpose!

Also we are v well aware that **defining** a new function does **not make the function run**. To **execute the function**, we need a **function call**. This is also known as a **function invocation**.

The way to invoke a function is to refer to it by name, followed by parentheses

```
def hello():
    print("Hello")
    print("Glad to meet you")

print(type(hello))
print(type("hello"))

hello()
print("Hey, that printed 2 lines with 1 line of code!")
hello()  # do it again, just because we can...
```

```
<class 'function'>
<class 'str'>
Hello
Glad to meet you
Hey, that printed 2 lines with 1 line of code!
Hello
Glad to meet you
```



Let's take a closer look at what happens when you define a function and when you execute the function.

Try mapping the above code and what is happening inside the memory.

- First, note that, when it executes line 1 which is reading the function definition, it does not execute lines 2 and 3. Instead, as you can see in **frames "Global frame"** area, it **creates a variable named hello** whose value is a **python function object**. In the diagram that object is labeled hello() with a notation above it that it is a function.
- Later, the next line of code to execute is line 5. Just to emphasize that hello is a variable like any other, and that functions are python objects like any other, just of a particular type, line 5 prints out the type of the object referred to by the variable hello. It's type is officially 'function'.Line 6 is just there to remind you of the difference between referring to the variable name (function name) hello and referring to the string "hello".
- After that interpreter gets to line 8, which has an invocation of the function. The way function invocation works is that the code block inside the function definition is executed in the usual way, but at the

end, execution jumps to the point after the function invocation.

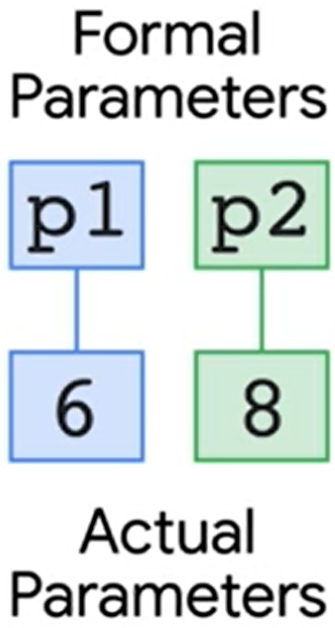
- during function call, we say that control has passed from the top-level program to the function hello after which it prints out two lines defined in the function and control will be passed back to the point after where the invocation was started.
- Same process will happen with another hello() invocation

Function Parameters

We know that function is a mechanism to do something over and over again. And you would have seen examples where a function was passed with some values called as parameters/arguments

These values, often called arguments or actual parameters or parameter values, are passed to the function by the user.

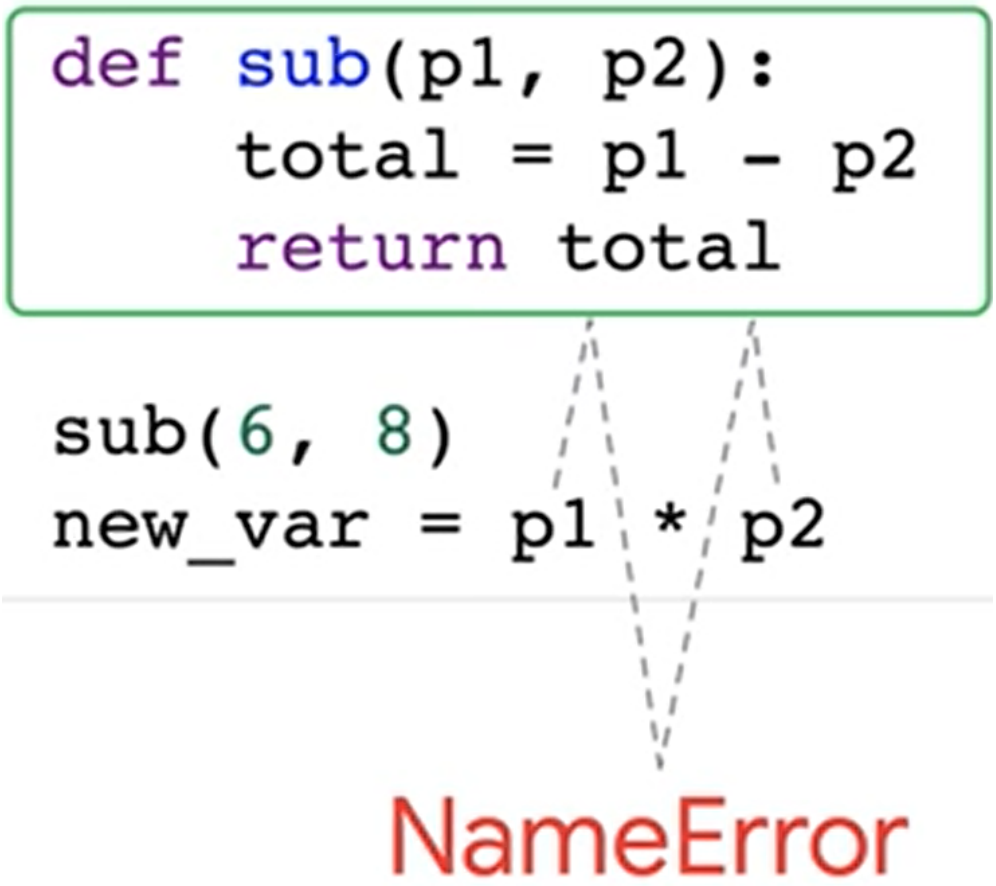
```
def sub(p1, p2):
    total = p1 - p2
    return total

sub(6, 8)
```



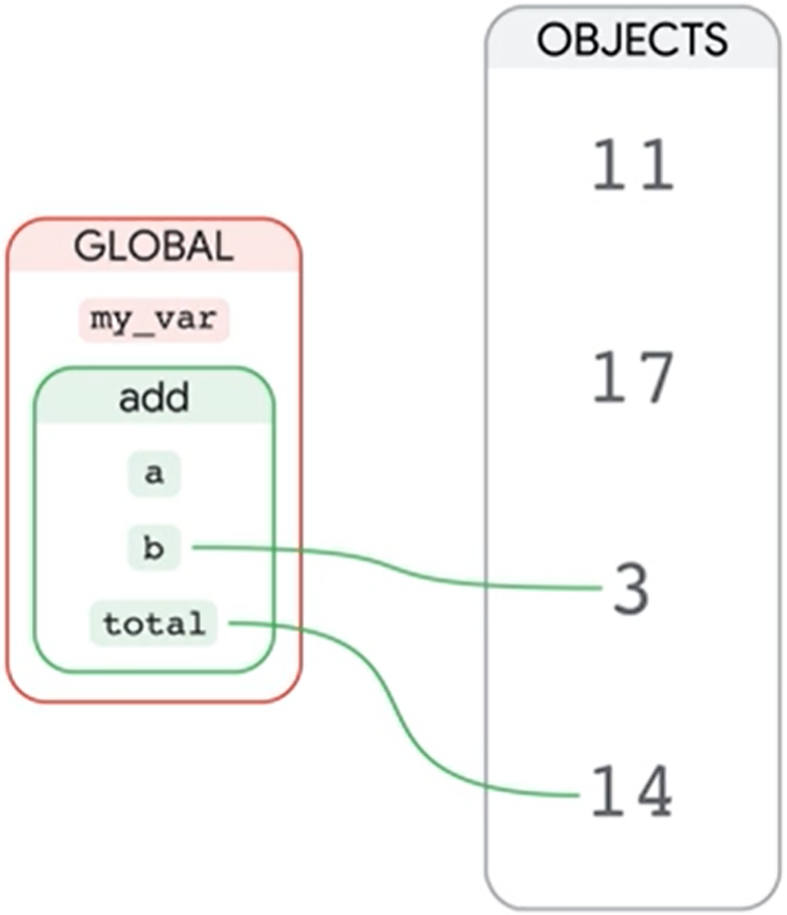
In above figure we can clearly see there are 2 different forms of parameters formal and Actual

When a function has one or more parameters, the names of the parameters appear in the function definition (in above image def sum(p1,p2)) and the values to assign to those parameters appear inside the parentheses of the function invocation.

But always remember the scope of variables defined as formal parameters which is p1, and p2 have their scope only inside the function. They can never be accessed outside in other function or as a global variable, if you try to do so , you will end up with errors.

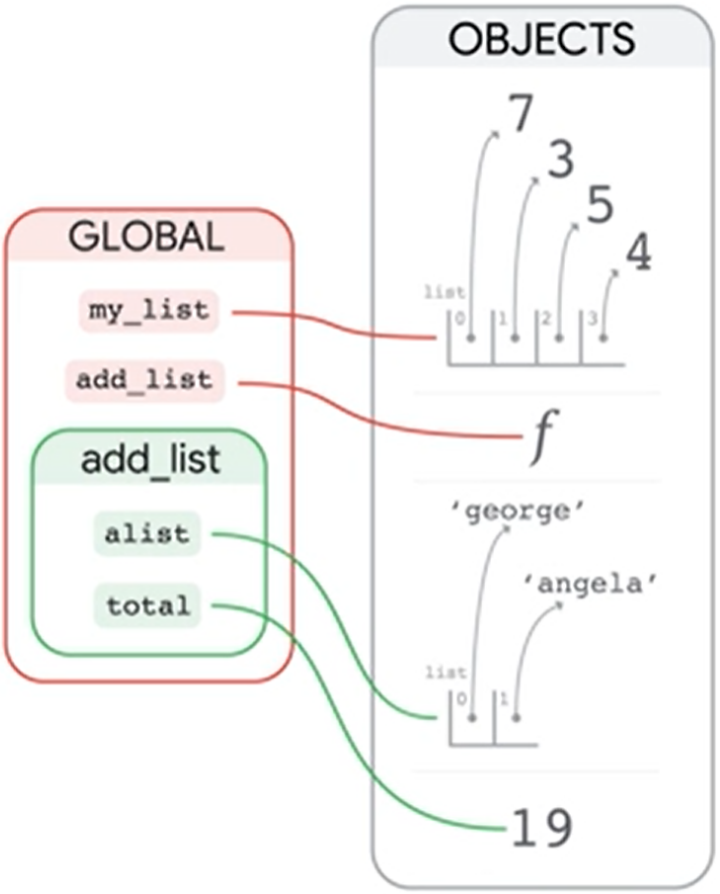


```
def add(a, b):
    total = a + b
    a = 17
    return total

my_var = 11
add(my_var, 3)
```



- When the interpreter started to run the program, it first read add function and created an add object.
- Later it went to my_var marked it as a global variable and assigned value 11
- add function was then called which had 2 arguments my_var,3 .
    - Firstly it assigned the value of 11 to my_var inside the argument list
    - And then passed actual arguments which are 11,3 to add function.
- Now the function executes , calculates total and passes value 17 to the function caller but wait , the value of a was changed to 17 inside the function and a is 11 which is my_var so will the value of my_var change too?
- Answer is no, it wont cause whatever happens inside a function with formal arguments does not change anything in the actual arguments provided the passed parameter is not mutable like lists/dictionary

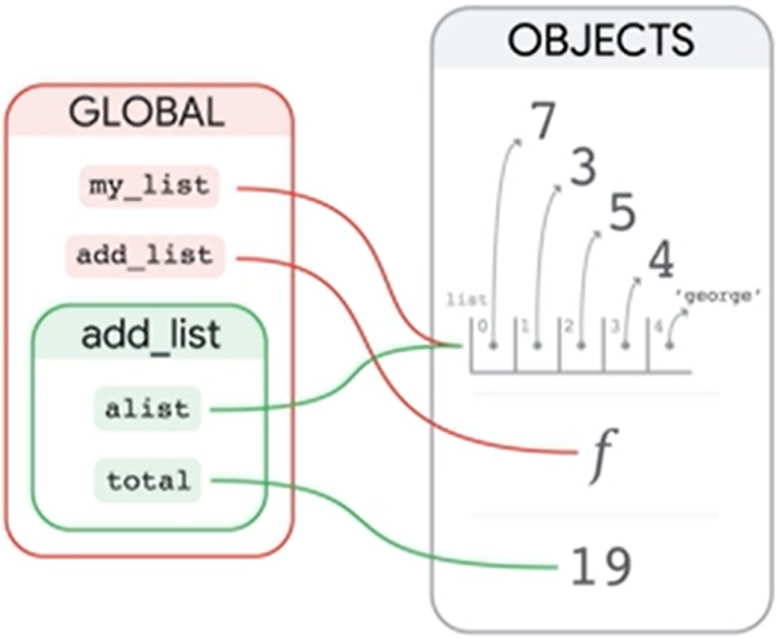
```
def add_list(alist):
    total = 0
    for num in alist:
        total += num
    alist = ['george', 'angela']
    return total

my_list = [7, 3, 5, 4]
add_list(my_list)
```



Let's see an example of Function with Lists

- In above program , definition of add_list was interpreted and an object of add_list was created.
- Later my_list var was initialized and it started referencing list of 4 integers present in the memory
- Now later my_list was passed as [7,3,5,4] which is the actual argument to add_list and alist is now pointing to a list of same what my_list is pointing.
- changing the value of alist inside the function will change anything to my_list?
- Answer is again no because now alist is assigned with a new list meaning alist is now pointing or referencing to a list which is present in some other part of the memory
- But let's see other possibilities.

```
def add_list(alist):
    total = 0
    for num in alist:
        total += num
    alist.append('george')
    return total

my_list = [7, 3, 5, 4]
add_list(my_list)
```



In here, we see that the alist is appending a value 'george', will that mean the last element of my_list too will become 'george'?

The answer is yes, because when you pass a list , you are just passing a reference point , which means alist and my_list both are pointing/referencing to a same memory block and append is adding the same list.

There are 2 kinds of functions that we will come across:

1. Procedure/Subroutine functions which will only perform a work, like drawing a structure, we don't expect anything in return from that function.
2. Fruitful function are the ones which returns as some relevant information after the work is done like if we need to check a credit card balance, wouldn't it look stupid if the function we write just goes in database , checks the balance and keeps it to itself? If a function returns the value it is a fruitful function

But what converts a procedure into --> fruitful function ??

return keyword and its implementation.

But remember all function will something, that's how the structure of a function is defined but if you are **not using return keyword explicitly** then **the the function returns none**

```
def add(1, 4):
        5

total = add(1, 4)
print(total)
```

```
None
```

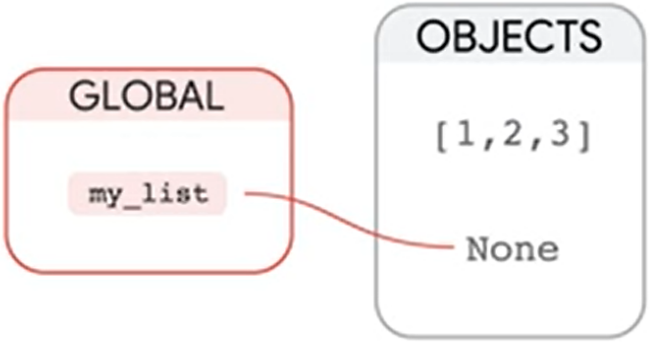
While learning lists we came across several methods while performing actions over a list among which append was one very useful method which helps us to add a new element inside the list.

But we saw that we cannot expect:
lst1= lst.append('test')
this to have lst1 as = [old_list_elements , test] because append is a method which performs operation over the main object and returns nothing so lst1 will have nothing though lst will have test appended.

which means append is procedure and not a fruitful function.

```
my_list = [1,2,3]
my_list = my_list.append(4)
print(my_list)
```

None



Another mistake that you might do is consider print and return as same.

print

```
def new_balance1(old_balance, new_charge):
    print(old_balance + new_charge)

print(new_balance1(2, 2))
```

4

None

---

return

```
def new_balance2(old_balance, new_charge):
    return old_balance + new_charge

print(new_balance2(2, 2))
```

4

- In above example when you use print, it will print the result to the standard output screen without passing anything but none to the calling function and hence as a result you would see 4 and None

Let's see another example in a little more detail:

```
def square(x):
    y = x * x
    return y

toSquare = 10
result = square(toSquare)
print("The result of {} squared is {}.".format(toSquare, result))

```

```
The result of 10 squared is 100.
```

The **return** statement is followed by an expression which is evaluated. Its result is returned to the caller as the “fruit” of calling this function.

Because the return statement can contain any Python expression we could have avoided **creating the temporary variable y** and simply used return x*x. Try modifying the square function above to see that this works just the same.

On the other hand, using **temporary variables like y** in the program above **makes debugging easier**. These temporary variables are referred to as **local variables**.

Notice something important here.

The name of the variable we pass as an argument — toSquare variable — has nothing to do with the name of the formal parameter which is x.

It is as if x = toSquare is executed when square is called. It doesn’t matter what the value was named in the caller (the place where the function was invoked). Inside square, it’s name is x. So as we learned anything happening with x will not affect toSquare as it is assigned to an int and int values are immutable.

```
def weird():
    print("here")
    return 5
    print("there")
    return 10

x = weird()
print(x)
```

```
here
5
```

Why using return at right place is very important?

Remember the moment interpreter reaches return, it takes the return value and comes out of the function without executing any more commands that would have been written in the function.

As we see in above example, it return 5 and then exited without executing "there" . It is a best practice and try to make it a habit of returning only one return to avoid confusions.

Try to write a program with below conditions:

- define a function called longer_than_five:
  - You'll want to pass in a list of strings (representing people's first names) to the function.
  - You'll want to iterate over all the items in the list, each of the strings.
  - As soon as you get to one name that is longer than five letters, you know the function should return True – yes, there is at least one name longer than five letters!
  - And if you go through the whole list and there was no name longer than five letters, then the function should return False.

Questions:

1. What is wrong with the following function definition:

```
def addEm(x, y, z):
    return x+y+z
    print('the answer is', x+y+z)
```

2. What will the following function return?

```
def addEm(x, y, z):
    print(x+y+z)
```

3. What will the following code output?

```
def square(x):
    y = x * x
    return y
print(square(5) + square(5))
```

4. Which will print out first, square, g, or a number?

```
def square(x):
    print("square")
    return x*x
def g(y):
    print("g")
    return y + 3
print(square(g(2)))
```

5.How many lines will the following code print?

```
def show_me_numbers(list_of_ints):
    print(10)
    print("Next we'll accumulate the sum")
    accum = 0
    for num in list_of_ints:
        accum = accum + num
        return accum
        print("All done with accumulation!")
show_me_numbers([4,2,3])
```

Decoding a Function

When you see a function definition you will try figure out what the function does, but, unless you are writing the function, you won't care how it does it.

For example, here is a summary of some functions we have seen already.

- **input** takes one parameter, a string. It is displayed to the user. Whatever the user types is returned, as a string.
- **int** takes one parameter. It can be of any type that can be converted into an integer, such as a floating point number or a string whose characters are all digits.

Sometimes, you will be presented with a function definition whose operation is not so neatly summarized as above. Sometimes you will need to look at the code, either the function definition or code that invokes the function, in order to figure out what it does.

To build your understanding of any function, you should aim to answer the following questions:

- How many parameters does it have?
- What is the type of values that will be passed when the function is invoked?
- What is the type of the return value that the function produces when it executes?

If you try to make use of functions, ones you write or that others write, without being able to answer these questions, you will find that your debugging sessions are long and painful.

Let's see some examples:

How many parameters does function cyu3 take?

```
def cyu3(x, y, z):
  if x - y > 0:
    return y -2
  else:
    z.append(y)
    return x + 3
```

What are the possible types of variables x and y?

```
def cyu3(x, y, z):
  if x - y > 0:
    return y -2
  else:
    z.append(y)
    return x + 3
```

What are the possible types of variable z?

```
def cyu3(x, y, z):
        if x - y > 0:
                return y -2
        else:
                z.append(y)
        return x + 3
```

What are the possible types of the return value from cyu3?

```
def cyu3(x, y, z):
  if x - y > 0:
    return y -2
  else:
    z.append(y)
    return x + 3
```

Accumulating Functions

We have used the len function a lot already. If it weren't part of python, our lives as programmers would have been a lot harder.

But hey, actually, not that much harder now that we know how to define functions, we could define len ourselves if it did not exist.

We have already seen a lot many examples how well we could use accumulator patterns to count the number of lines in a file or accumulate the number of elements present in a sequence or summation of all the elements of a sequence.

Let's use that same idea and wrap it in a function definition. We'll call it **mylen** to distinguish it from the real len which already exists. We actually could call it len, but that wouldn't be a very good idea, because it would replace the original len function, and our implementation may not be a very good one.

```
1  def mylen(seq):
2      c = 0 # initialize count variable to 0
3      for _ in seq:
4          c = c + 1    # increment the counter for each item in seq
5      return c
6
7  print(mylen("hello"))
8  print(mylen([1, 2, 7]))
```

```
5
3
```

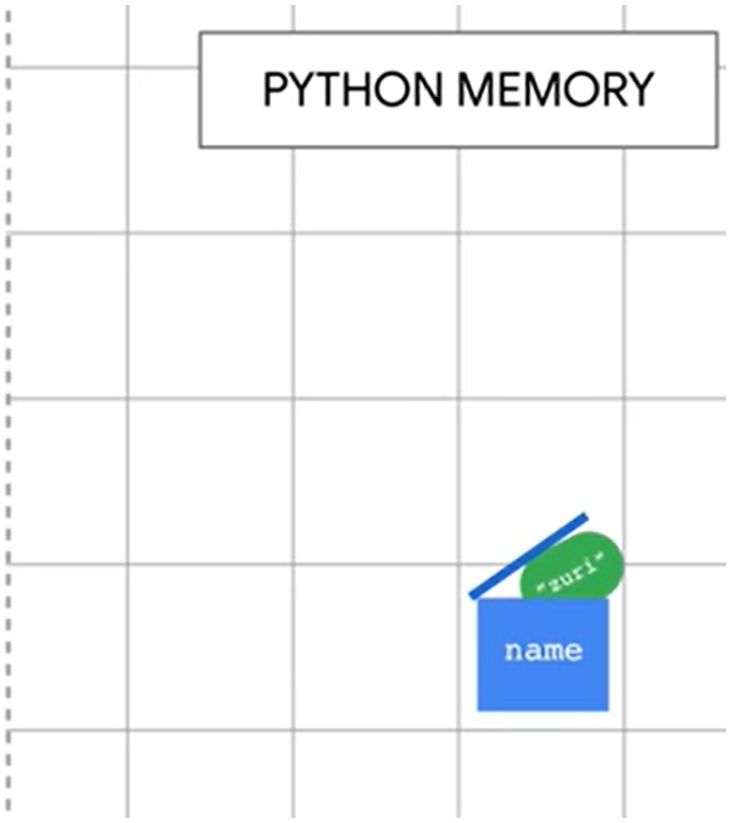
Variables and parameters are local

We have used so many variables so far, we have learned when to use them how to use them, how to name , how to assign and reassign them but there are several other questions that might arise in your head when you include a variable in a program like:

- Where do those variables live? Meaning stored
- How does the program find them when it needs them?

```
name = "zuri"
print(name)
```

"zuri"



There are several set of rules that define this part of memory allocation which are called as Scope.

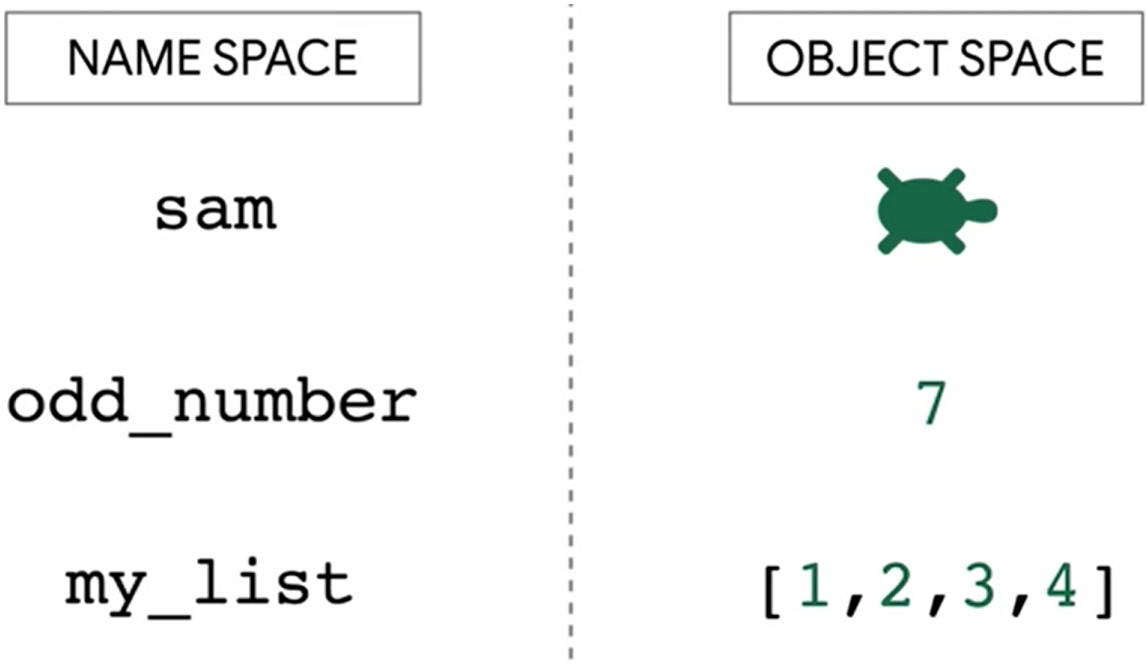
Scope in short is the block of code where variables can be accessed.

You need to understand this term scope in a better clear way, or else you would end up creating lot many confusing errors and wasting lot of time

```
a = 14
def add(p1,p2):
    b = p1 + p2

add(7,3)
print(b)
```

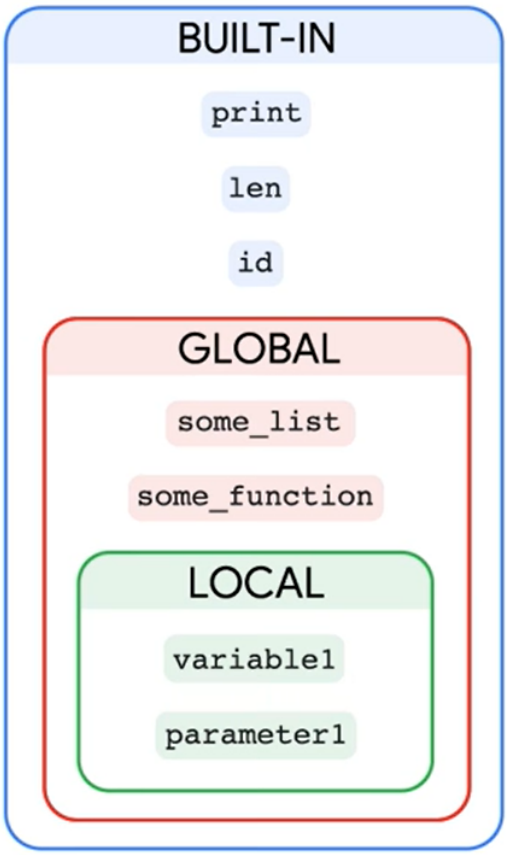
```
NameError:
name 'b' is not
defined on line 6
```

Python universe can be divided into 2 space:

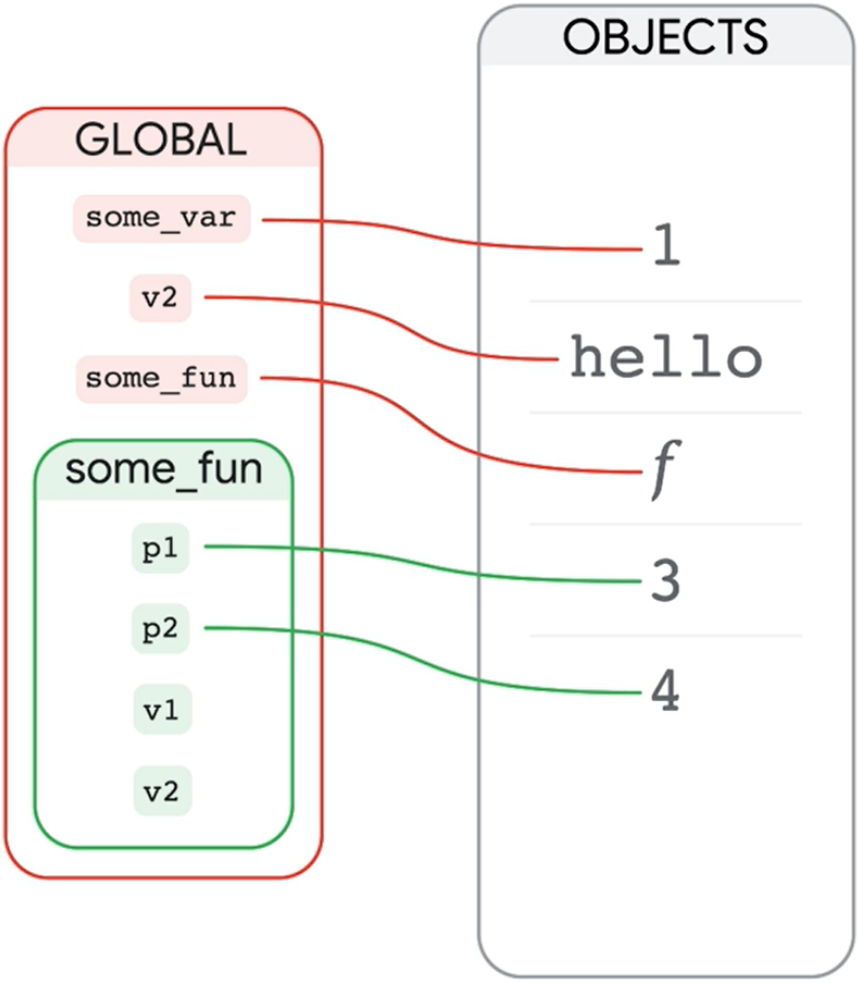


All objects are created in object space.
But Name space has further 3 categories:

- BUILT-IN is the place where all pre-defined functions exist.
- Global are the ones which are can be any list, or functions that we will define
- Local scope is the place where all the variables/parameters exist



```
some_var = 1
v2 = "hello"

def some_fun(p1, p2):
    v1 = some_var + p1 + p2
    print(v1)
    v2 = 3 + 3

some_fun(3, 4)
```



Let's try to understand the above example:

- Global scope has at the very first place 2 variables assigned with their values.
- Then some_fun is created in Global scope
- And then the functions call some_fun is interpreted.
    - First thing that happens is we a local space of some_fun is created , in that namespace we have p1, p2, v1,v2 variables created. Your question might be why do we have v1 and v2 created in the local space when we have yet not used v1 and v2?
        - Because any variable which is assigned in a function gets space in local namespace right away irrespective of any condition.
    - then v1=some_var + p1 + p2 is exectued but hey , some_var does not exist in local space.

In this case it will leave local space and go searching for it in Global space.

- Similarly, when it comes to next line print(v1), print definition does not exist in local so it searches in global and then BUILT-IN.

This is how the searching of all elements in a python programming looks like.

```
1 def square(x):
2     y = x * x
3     print(y)
4     return y
5
6 z = square(10)
7 print(y)
```

```
100
0
```

For example in above program, in earlier version calling print(y) no line 7 would have given an error

```
NameError: name 'y' is not defined on line 6
```

Though in latest 3.8.5 this returns 0. But why would the error occur?

The variable y only exists while the function is being executed — **we call this its lifetime**. When the execution of the function terminates (returns), the local variables are destroyed.
And y is not defined in global scope so we should not expect it to return 100, that will be a really bad understanding of scope

Question:
What is the result of the following code?

```
def adding(x):
   y = 3
   z = y + x + x
   return z
def producing(x):
   z = x * y
   return z
print(producing(adding(4)))
```

Global Variables

Variable names that are at the top-level, not inside any function definition, are called global.

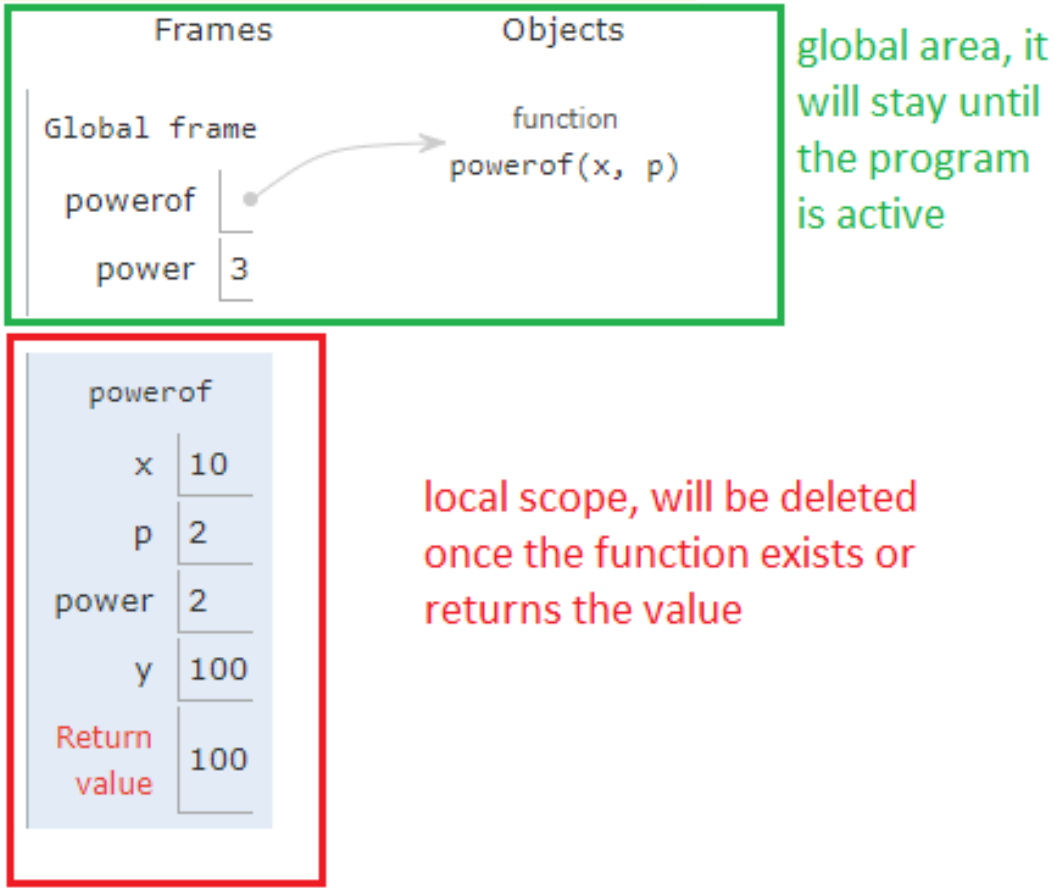
Though it is legal for a function to access a global variable. However, this is considered bad form by nearly all programmers and should be avoided.

```
1  def badsquare(x):
2      y = x ** power
3      return y
4
5  power = 2
6  result = badsquare(10)
7  print(result)
```

```
100
```

Although the badsquare function works, it is **silly and poorly written**.

- As we have already went through the flow how python works, Firstly, Python looks at the variables that are defined as local variables in the function. We call this the **local scope**.
- If the variable name is not found in the local scope, then Python looks at the global variables, or global scope. which is the case in the code above.
- power is not found locally in badsquare but it does exist globally. The appropriate way to write this function would be to pass power as a parameter



```
def powerof(x,p):
    power = p   # Another dumb mistake
    y = x ** power
    return y

power = 3
result = powerof(10,2)
print(result)
```

```
100
```

The **value of power** in the **local scope** was **different** than the **global scope**. That is because in this example power was used on the left hand side of the assignment statement power = p.

When a **variable name** is used on the **left hand side of an assignment** statement **Python creates a local variable**. When a local variable has the same name as a global variable we say that the **local shadows the global**.

A **shadow** means that the **global variable cannot be accessed by Python** because the local variable will be found first. This is another good reason not to use global variables. As you can see, it makes your code confusing and difficult to understand.

Functions calling other functions (Composition)

It is important to understand that each of the functions we write can be used and called from other functions we write. This is one of the most important ways that computer programmers take a large problem and break it down into a group of smaller problems. **This process of breaking a problem into smaller subproblems is called functional decomposition**.

Let's see a simple example:

```
def square(x):
    return x * x

def sum_of_square(x,y,z):
    return square(x) + square(y) + square(z)

a=-5
b=2
c=10
print(sum_of_square(a,b,c))
```

```
129
```

Even though this is a pretty simple idea, in practice this example illustrates many very important Python concepts, including local and global variables along with parameter passing.

Note that the body of square is not executed until it is called from inside the sum_of_squares function for the first time

there are two groups of local variables, one for square and one for sum_of_squares. Each group of local variables is called a stack frame. The variables x is local variables in both functions. This is completely different variables, even though they have the same name.

Each function invocation creates a new frame, and variables are looked up in that frame.

```python
def most_common_letter(s):
    return best_key(count_freq(s))

def count_freq(st):
    d={}
    for i in st:
        if i not in d:
            d[i]=0
        d[i] = d[i] + 1
    return d

def best_key(d):
    key=list(d.keys())[0]
    for i in d:
        if d[i] > d[key]:
            key=i
    return key

print(most_common_letter("aaabddddddd"))
```

```
d
```

We have seen in earlier section how to see find the best key in a dictionary we tried here to recreate the same using functions.

But above program is a really bad way to write a program, as it is confusing.

- Variables used does not tell us what they will do, we need to understand the name of variables should give the reader an Idea about what the program will do.

Let's try to fix this and write this program again

```python
def most_common_letter(s):
    return best_key(count_freq(s))

def count_freq(st):
    d={}
    for i in st:
        if i not in d:
            d[i]=0
        d[i] = d[i] + 1
    return d

def best_key(d):
    key=list(d.keys())[0]
    for i in d:
        if d[i] > d[key]:
            key=i
    return key

print(most_common_letter("adadadaaaddddddddddd"))
```

```
d
```

This one looks pretty good.

Consider the following Python code. What will it return

```
def pow(b, p):
   y = b ** p
   return y
def square(x):
   a = pow(x, 2)
   return a
n = 5
result = square(n)
print(result)
```

Print vs Return

We saw earlier as well how wrong understanding between the difference of print and return could lead to bad programming.

The print statement is fairly easy to understand. It takes a python object and outputs a printed representation of it in the output window. You can think of the print statement as something that takes an object from the land of the program and makes it visible to the land of the human observer.

Print is for people. Remember that slogan. Printing has no effect on the ongoing execution of a program. It doesn't assign a value to a variable. It doesn't return a value from a function call.

Passing Mutable Objects

when a function (or method) is invoked and a parameter value is provided, a new stack frame is created, and the parameter name is

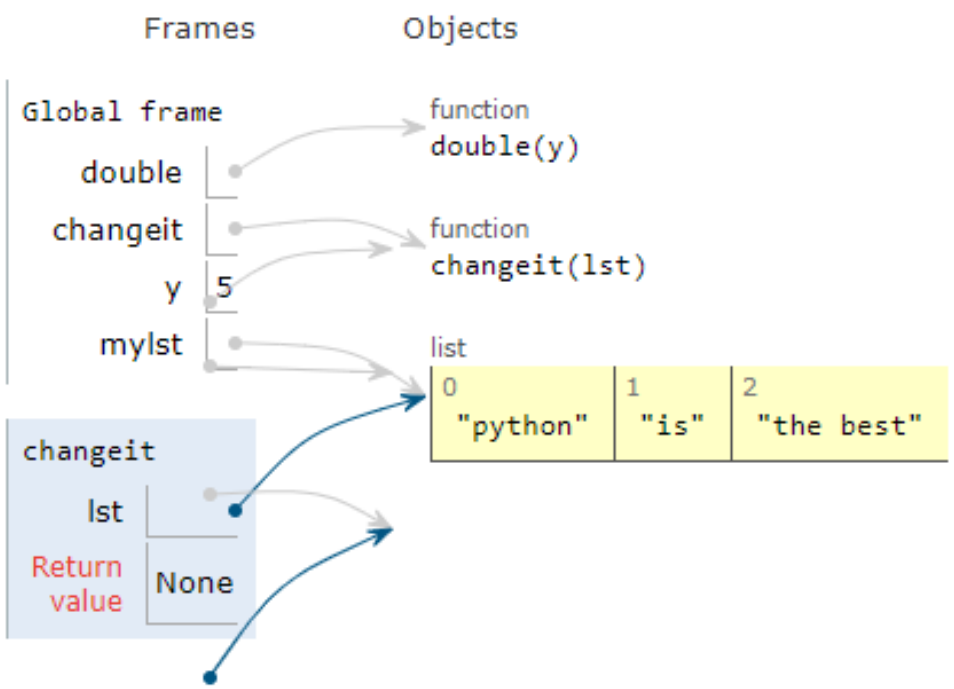
bound to the parameter value. What happens when the value that is provided is a mutable object, like a list or dictionary? Is the parameter name bound to a copy of the original object, or does it become an alias for exactly that object?

In python, the answer is that it becomes an alias for the original object.

This answer matters when the code block inside the function definition causes some change to be made to the object (e.g., adding a key-value pair to a dictionary or appending to a list).

```
1  def double(y):
2      return y * 2
3  def changeit(lst):
4      lst[0]="python"
5      lst[2]="the best"
6
7  y=5
8  double(y)
9  print(y)
10
11 mylst=["Java","is","good"]
12 changeit(mylst)
13 print(mylst)
```

```
5
['python', 'is', 'the best']
```



running double does not change the global y. But running changeit does change mylst because the 2 variable lst, mylst are pointing/referring to same list

Side effects of global Variable

```
def double(n):
    global y  #we are telling the interpreter
              #to consider y as global variable
              #meaning any change inside the function
              #would also reflect in the global scope
              #as you saw in the output
    y = 2 * n

y = 5
double(y)
print(y)
```

```
10
```

This example might be as per your expectation but when you will write bigger programs, the side effects of writing program in this way can be very difficult to debug.

```
def changeit(lst):
    lst[0] = "python"
    lst[2] = "the best"
    return lst

mylst = ['Java', 'is', 'good']
newlst = changeit(list(mylst))#this is the important part
                              #which is preventing the side effect
                              #we have while passing sequences/maps
print(mylst)
print(newlst)
```

```
['Java', 'is', 'good']
['python', 'is', 'the best']
```

Although Python provides us with many list methods, it is good practice and very instructive to think about how they are implemented. Implement a Python function that works like the following:

- count
- in
- reverse
- index
- insert

```
def count(lst,count_element):
    accum=0
    for i in lst:
        if i == count_element:
            accum= accum+1
    return accum

def in_exist(lst,element):
    for i in lst:
        if i == element:
            return True
        else:
            return False

def reverse(lst):
    rev=[]
    for i in lst:
        rev= [str(i)] + rev
    return rev

def index(lst,element):
    index=0
    for i in lst:
        index =index +1
        if i == element:
            return index -1

def insert(lst,element,pos):
    if len(lst) > pos + 1:
        return lst[:pos] + [element] + lst[pos:]
```

```
lst=[1,23,4,5,'arshad','shaikh','arshad',1,2,11,1]
print(count(lst,1))
print(in_exist(lst,6))
print(reverse(lst))
print(index(lst,1))
print(lst)
print(insert(lst,3,3))
```

Write a function replace(s, old, new) that replaces all occurences of old with new in a string s:

test(replace('Mississippi', 'i', 'I'), 'MIssIssIppI')

```
def replace(s, old, new):
    # your code here
    rep=""
    for i in s:
        if i == old:
            rep= rep + new
        else:
            rep= rep + i
    return rep
```

Write a Python function that will take a the list of 100 random integers between 0 and 1000 and return the maximum value. (Note: there is a builtin function named max but pretend you cannot use it.)

```
import random
def ran():
    lst=[]
    for i in range(100):
        lst.append(random.randrange(1,1001))
    return lst

def max_value(lst):
    maxi=lst[0]
    for i in lst:
        if i > maxi:
            maxi=i
    return maxi

print(max_value(ran()))
```

Write a function findHypot. The function will be given the length of two sides of a right-angled triangle and it should return the length of the hypotenuse. (Hint: x ** 0.5 will return the square root, or use sqrt from the math module)

```
def findHypot(a,b):
    # your code here
   return ((a*a) + (b*b))** 0.5
```

PART 7