**Functions**

**Introduction to Functions**

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

**So what is a function?**

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design

**def Statements**

Let's see how to build out a function's syntax in Python. It has the following form:

**in** other lang

​

function declaration

function defination

function call

int func(int a , int b);

​

int func(int a,int b)

{

}

func(20,40)

​

​

​

**def** name\_of\_function(arg1 ,arg2): *#function defination - formal arguments*

'''

This is where the function's Document String (docstring) goes

function:

parameters:

return value:

optional:

'''

*# Do stuff here*

*# Return desired result*

**return** x,y,c *#pack*

​

t **=** name\_of\_fun()

x,y,z **=** t

​

t **=** name\_of\_function(10,20) *#function call ,actual arguments*

x,y,z **=** t *#unpack*

​

lst **=**list()

tuple()

​

lst.append()

​

sorted(tup) **-->**funtion

​



We begin with <code>def</code> then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a [built-in function in Python](https://docs.python.org/2/library/functions.html) (such as len).

​

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

​

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *\*whitespace\** to organize code. Lots of other programing languages do not do this, so keep that in mind.

​

Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

​

After all this you begin writing the code you wish to execute.

​

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

​

When a function is defined , it will have parameters and these parameters are useful to receive the values from outside the function , these are called as "formal arguments"

When we call a function we should pass data or values to the functions then these arguments are called as "actual arguments"

Type *Markdown* and LaTeX: *α*2

**def** say\_hello():

print('hello')

say\_hello()

print(say\_hello)

hello

<function say\_hello at 0x063414F8>

Call the function:

print(say\_hello())

​

say\_hello()

hello

None

hello

None

hello

Type *Markdown* and LaTeX: *α*2

**def** greeting(name):

print('Hello %s' **%**(name))

greeting('Jose')

greeting(10)

Hello Jose

**Using return**

Let's see some example that use a return statement. return allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

**def** add\_num(num1,num2):

**return** num1**+**num2

print(add\_num(4,5))

9

*# Can also save as variable due to return*

result **=** add\_num(4,5)

print(result)

9

What happens if we input two strings?

add\_num('one','two')

Out[74]:

'onetwo'

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using break, continue, and pass statements in our code.

We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

**def** is\_prime(num):

'''

Naive method of checking for primes for the input number.

'''

**for** n **in** range(2,num):

**if** num **%** n **==** 0:

print(num,'is not prime')

**break**

**else**: *# If never mod zero, then prime*

print(num,'is prime!')

is\_prime(16)

print(is\_prime.\_\_doc\_\_)

16 is not prime

print(is\_prime.\_\_doc\_\_) *# to print the api documentation using doc string \_\_doc\_\_- dunder methods or special methods*

Naive method of checking for primes for the input number.

is\_prime(17)

17 is prime!

Note how the else lines up under for and not if. This is because we want the for loop to exhaust all possibilities in the range before printing our number is prime.

Also note how we break the code after the first print statement. As soon as we determine that a number is not prime we break out of the for loop.

We can actually improve this function by only checking to the square root of the target number, and by disregarding all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

*# import math*

**from** math **import** sqrt

​

*#include <stdio.h>*

*#import java.util.io*

​

**def** is\_prime2(num):

'''

Better method of checking for primes.

'''

**if** num **%** 2 **==** 0 **and** num **>** 2:

**return** **False**

**for** i **in** range(3, int(sqrt(num)) **+** 1, 2):

**if** num **%** i **==** 0:

**return** **False**

**return** **True**

*# is\_prime2(18)*

is\_prime2(18)

Out[14]:

False

##Aliasing function names Python has a special feature where you can create an alias of any function, i.e. if you think a function name is too long and not worth typing every time, you can decide a new name for it, without altering the original function. Taking the example of add() function, suppose we want a shorter or different name for this function, but without editing the original function, we can do so by using the function aliasing feature. Suppose we want to rename is\_prime2() to a(), then

a **=** is\_prime2

​

​

​

​

a(18)

Out[18]:

False

a **=**20

b **=** a

Type *Markdown* and LaTeX: *α*2

Type *Markdown* and LaTeX: *α*2

*#retruning multiple results*

**def** sum(a,b):

c **=** a **+** b

d **=** a **-** b

**return** c,d

​

​

t**=** sum(10,20) *#packing*

x,y **=** t *#tuple unpacking*

print("Addition value is {},Subtraction value is {}".format(x,y))

​

Addition value is 30,Subtraction value is -10

*# Pass by value or call by value and Pass by reference or call by reference neither of these 2 are applicable in python*

*# The values are sent to functions by means of object references*

*# immutable or non modifiable*

*# mutable or modifiable*

*# if new mutable object is created then it create new memory*

*# Passing an integer to a function*

**def** modify(x):

print(x)

print(id(x))

x**=**15

*#when new object 15 is created then x is tageed to 15 instead of 10 , but without creating if we print x then it will be 10 only*

print(x,id(x))

​

x **=** 10

print(id(x))

modify(x)

print(x,id(x))

1658807536

10

1658807536

15 1658807616

10 1658807536

*#passing a program to pass a list and modify it*

**def** modify(lst):

print(lst,id(lst))

*# lst.append(9) # but if new object inside the function is created then it will not be modified outside*

lst **=** [10,11,12]

print(lst,id(lst))

lst **=** [1,2,3,4]

print(lst,id(lst))

modify(lst)

print(lst,id(lst))

​

[1, 2, 3, 4] 100961096

[1, 2, 3, 4] 100961096

[10, 11, 12] 98458320

[1, 2, 3, 4] 100961096

The actual arguments used in a function call are of 4 types

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. variable length arguments

*#positional arguments*

**def** attach(s1,s2):

s3 **=** s1**+**s2

print('Total string is' **+**s3)

attach('New','york')

​

Total string isNewyork

attach('york','new') *# since positional values can not be interchanged*

Total string isyorknew

*# Keyword Arguments or named arguments*

**def** grocery(item,price):

print('item = %10s' **%** item)

print('price = %10.2f' **%** price)

​

grocery(item **=** 'sugar',price **=** 50)

grocery(price**=**50,item **=**'sugar')

item = sugar

price = 50.00

item = sugar

price = 50.00

*#default Arguments*

**def** grocery(item,price**=**60):

print('item = %10s' **%** item)

print('price = %10.2f' **%** price)

grocery(item **=** 'sugar')

grocery(price**=**50,item **=**'sugar')

item = sugar

price = 60.00

item = sugar

price = 50.00

*#variable length argument or formal arguments*

**def** add(farg,**\***args):

print('formal arguments =',farg)

sum **=**0

**for** i **in** args:

sum**+=**i

print('sum of all number =',(farg**+**sum))

add(5,10,20,30)

formal arguments = 5

sum of all number = 65

add(5,10,20,30)

formal arguments = 5

sum of all number = 65

# \*args and \*\*kwargs

Work with Python long enough, and eventually you will encounter \*args and \*\*kwargs. These strange terms show up as parameters in function definitions.

**def** myfunc(a**=**1,b**=**2,c**=**3,d**=**4,x**=**6,r**=**5,u**=**7,e**=**8):

**return** sum((a,b))**\***.05

​

lst**=** [1,2,3]

myfunc()

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-5-1d96558f1b65>** in <module>

2 **return** sum**((**a**,**b**))\*.05**

3

**----> 4** myfunc**(40,60,50,70,80)**

**TypeError**: myfunc() takes 4 positional arguments but 5 were given

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are positional arguments; that is, 40 is assigned to **a** because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the sum() function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a lot of parameters, and give each one a default value.

**def** myfunc(a**=**0,b**=**0,c**=**0,d**=**0,e**=**0):

**return** sum((a,b,c,d,e))**\***.05

​

myfunc(40,60,20)

Obviously this is not a very efficient solution, and that's where \*args comes in.

## \*args

When a function parameter starts with an asterisk, it allows for an arbitrary number of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

**def** myfunc(**\***args):

**for** i **in** args:

print(i)

lst **=** [1,2,3]

myfunc(lst)

myfunc(10,20,30,40,50,70,80,'strval',lst)

myfunc(10,20)

*# myfunc(10)*

​

[1, 2, 3]

10

20

30

40

50

70

80

strval

[1, 2, 3]

10

20

**def** myfunc(**\***args):

**for** i **in** args[8]: *#access only the eighth element in the list*

print(i)

lst **=** [1,2,3]

myfunc(10,20,30,40,50,70,80,'strval',lst)

​

​

1

2

3

**def** myfunc(**\***args):

lst **=**[1,2,30]

x **=** sum(lst) *#sum function takes only the iterable object and not just literal*

*# x = sum(1,2,3) #is error as the values are not iterable*

print(x)

**return** sum(args)**\***.05 *#here sum itself is an iterable function , so it iterates and takes the value indiviually and sums it up*

​

myfunc(40,60,20)

33

Out[7]:

6.0

Notice how passing the keyword "args" into the sum() function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

**def** myfunc(**\***spam):

**return** sum(spam)**\***.05

​

myfunc(40,60,20)

Out[15]:

6.0

​

## \*\*kwargs

Similarly, Python offers a way to handle arbitrary numbers of keyworded arguments. Instead of creating a tuple of values, \*\*kwargs builds a dictionary of key/value pairs. For example:

**def** myfunc(**\*\***kwargs):

**for** key **in** kwargs.keys():

print(key,"-",kwargs[key])

**for** val **in** kwargs.values():

print(val)

**for** both **in** kwargs.items():

print(both)

myfunc(fruit **=** 'pineapple',vegetable **=**'potato',cloth **=**'stain')

fruit - pineapple

vegetable - potato

cloth - stain

pineapple

potato

stain

('fruit', 'pineapple')

('vegetable', 'potato')

('cloth', 'stain')

**def** myfunc(**\*\***kwargs):

print(sum(kwargs.values()))

myfunc(a**=**1,b**=**2,c**=**3)

6

**def** myfunc(**\*\***kwargs):

**if** 'fruit' **in** kwargs:

print(f"My favorite fruit is {kwargs['fruit']}") *# review String Formatting and f-strings if this syntax is unfamiliar*

**else**:

print("I don't like fruit")

myfunc(fruit**=**'pineapple')

My favorite fruit is pineapple

*#myfunc()*

"Asha is a girl"

str.split()

['asha','is','a','girl']

'-'.join() **-** asha**-is** **-**a**-** girl

args **=** [eggs,spam]

eggs **and** spam **and** legs..

## \*args and \*\*kwargs combined

You can pass \*args and \*\*kwargs into the same function, but \*args have to appear before \*\*kwargs

**def** myfunc(**\***args,**\*\***kwargs):

**if** 'fruit' **and** 'juice' **in** kwargs:

print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}")

print(f"May I have some {kwargs['juice']} juice?")

**else**:

**pass**

myfunc('bananas','potatochips',fruit**=**'cherries',juice**=**'orange',snack**=**'chips')

I like bananas and my favorite fruit is cherries

May I have some orange juice?

Placing keyworded arguments ahead of positional arguments raises an exception:

myfunc(fruit**=**'cherries',juice**=**'orange','eggs','spam')

**File "<ipython-input-8-fc6ff65addcc>", line 1**

**myfunc(fruit='cherries',juice='orange','eggs','spam')**

**^**

**SyntaxError:** positional argument follows keyword argument

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.