Lecture 7 Functions and Parameters

IDS 400

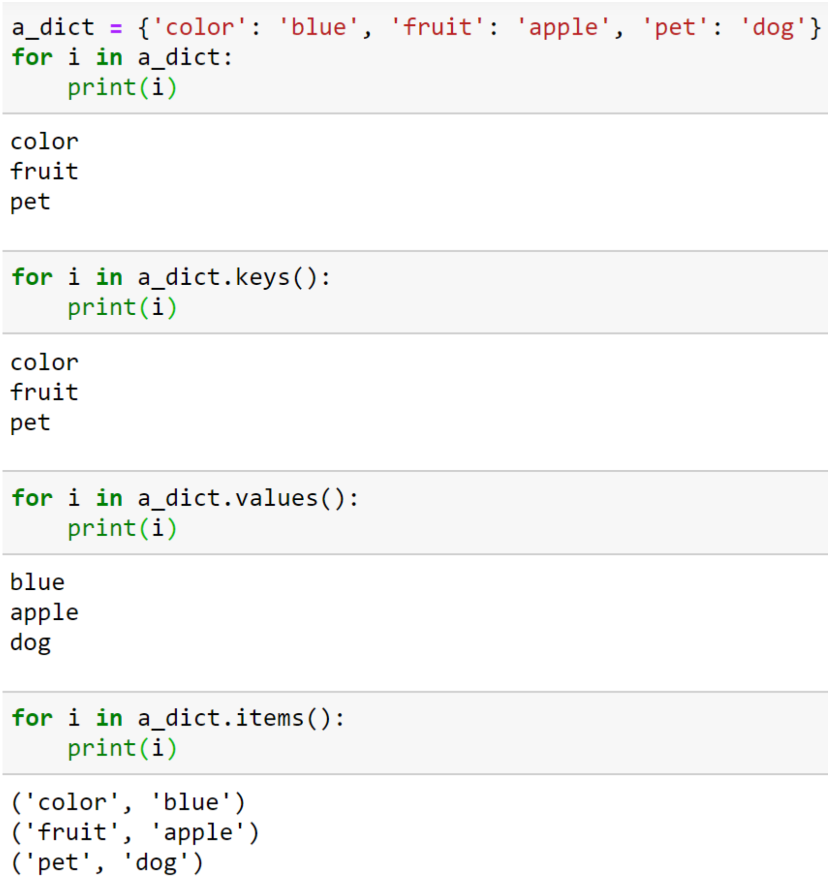
Programming for Data Science in Business

Tentative schedule

Date	Lecture Number	Topics
08/24	Lecture 1	Introduction
08/31	Lecture 2	Basic
09/07	Lecture 3	Condition
09/14	Lecture 4	Loop
09/21	Lecture 5	String + **Quiz 1**
09/28	Lecture 6	Type
10/05	Lecture 7	Function
10/12	Lecture 8	File + **Quiz 2**
10/19	Lecture 9	Pandas
10/26	Lecture 10	Numpy
11/02	Lecture 11	Machine Learning
11/09	Lecture 12	Visualization
11/16	Lecture 13	Web Scraping & Deep Learning
11/23	***Thanksgiving***	***No lecture***
11/30	**Final presentation**	In class presentation
12/05	**Project submission due**	

Iterate dictionaries

- Iterating through **Keys** directly.
- Iterating through **.Keys()**.
- Iterating through **.values()**.
- Iterating through **.items()**.



```
a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
for i in a_dict:
    print(i)
```

```
color
fruit
pet
```

```
for i in a_dict.keys():
    print(i)
```

```
color
fruit
pet
```

```
for i in a_dict.values():
    print(i)
```

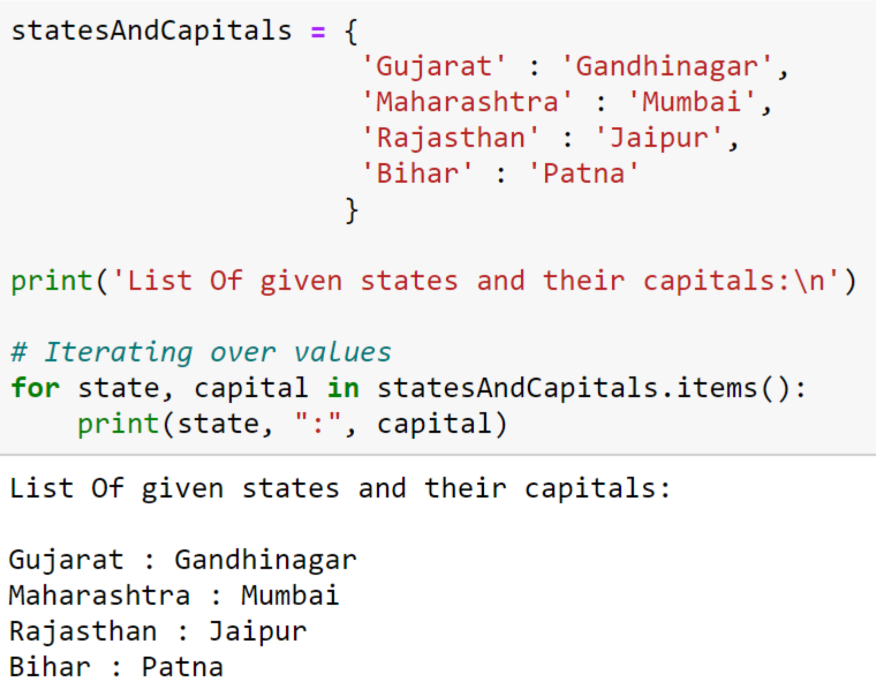
```
blue
apple
dog
```

```
for i in a_dict.items():
    print(i)
```

```
('color', 'blue')
('fruit', 'apple')
('pet', 'dog')
```

Iterate dictionaries

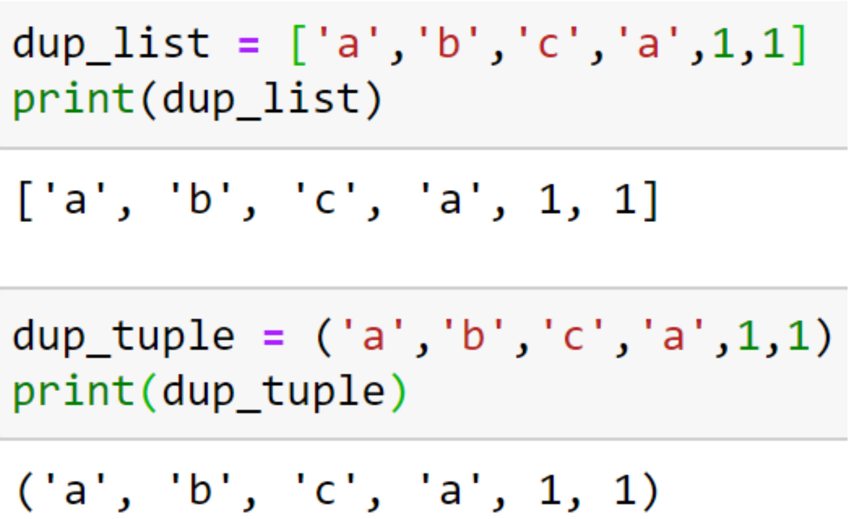
- Two iteration variables
  - We loop through the key-value pairs in a dictionary using two iteration variables.
  - Each iteration, the first variable is the key, and the second variable is the corresponding value for the key.



```
statesAndCapitals = {
                     'Gujarat' : 'Gandhinagar',
                     'Maharashtra' : 'Mumbai',
                     'Rajasthan' : 'Jaipur',
                     'Bihar' : 'Patna'
                    }

print('List Of given states and their capitals:\n')

# Iterating over values
for state, capital in statesAndCapitals.items():
    print(state, ":", capital)
```

```
List Of given states and their capitals:

Gujarat : Gandhinagar
Maharashtra : Mumbai
Rajasthan : Jaipur
Bihar : Patna
```

Duplicate elements



```
dup_list = ['a','b','c','a',1,1]
print(dup_list)
```

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

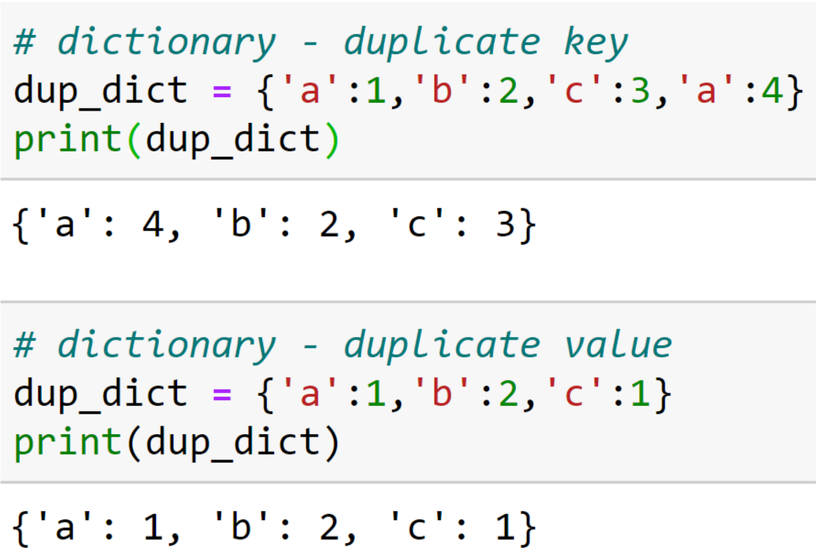
```
dup_tuple = ('a','b','c','a',1,1)
print(dup_tuple)
```

```
('a', 'b', 'c', 'a', 1, 1)
```

- In lists and tuples, duplicate elements are supported.

Duplicate elements

- Dictionaries do not support duplicate keys. However, we can have duplicate values in a dictionary.



```
# dictionary - duplicate key
dup_dict = {'a':1,'b':2,'c':3,'a':4}
print(dup_dict)
```

```
{'a': 4, 'b': 2, 'c': 3}
```

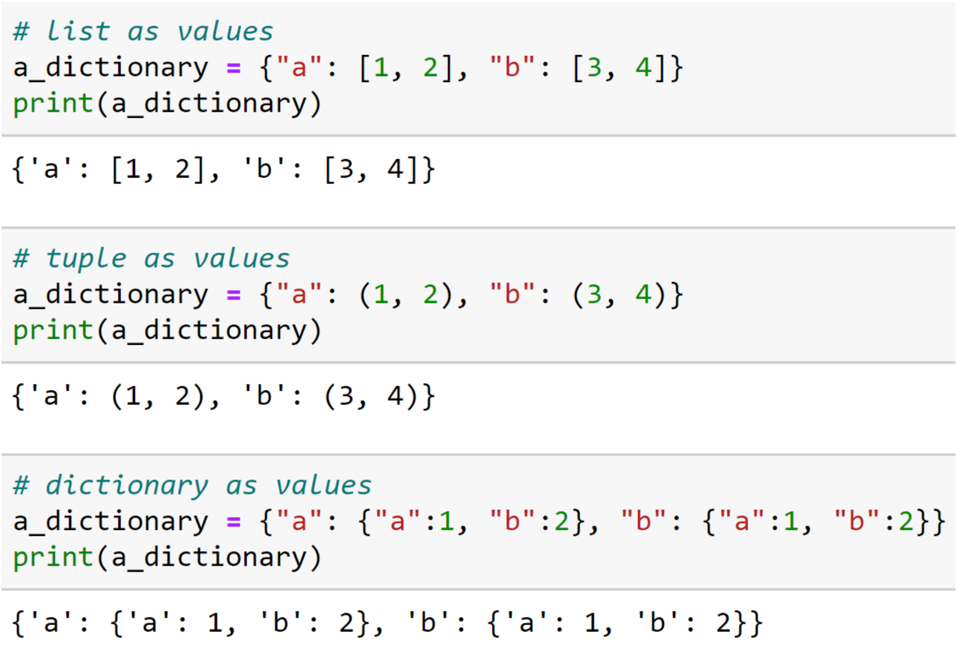
```
# dictionary - duplicate value
dup_dict = {'a':1,'b':2,'c':1}
print(dup_dict)
```

```
{'a': 1, 'b': 2, 'c': 1}
```

Containers as dictionary values

- More than one value can correspond to a single key using a list.

For example, with the dictionary {"a": [1, 2]}, 1 and 2 are both connected to the key "a"



```
# list as values
a_dictionary = {"a": [1, 2], "b": [3, 4]}
print(a_dictionary)
```

```
{'a': [1, 2], 'b': [3, 4]}
```

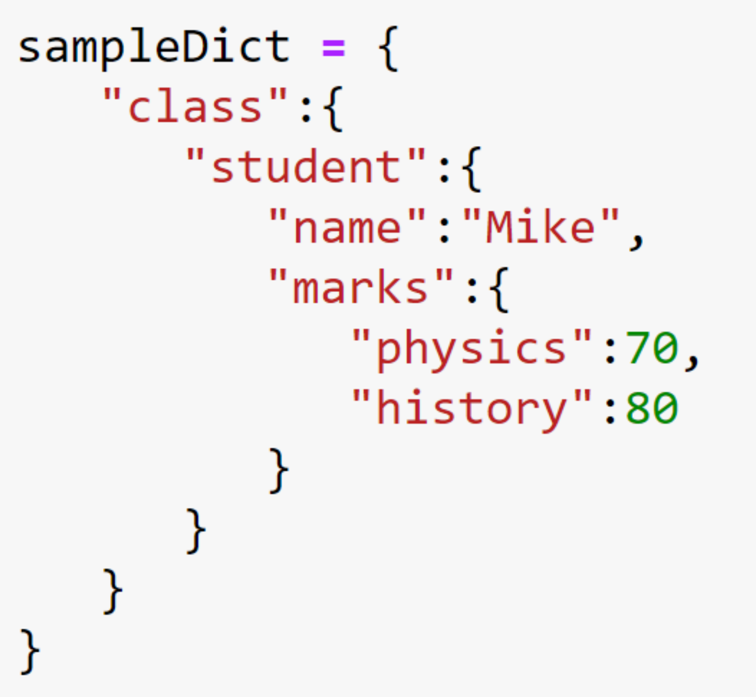
```
# tuple as values
a_dictionary = {"a": (1, 2), "b": (3, 4)}
print(a_dictionary)
```

```
{'a': (1, 2), 'b': (3, 4)}
```

```
# dictionary as values
a_dictionary = {"a": {"a":1, "b":2}, "b": {"a":1, "b":2}}
print(a_dictionary)
```

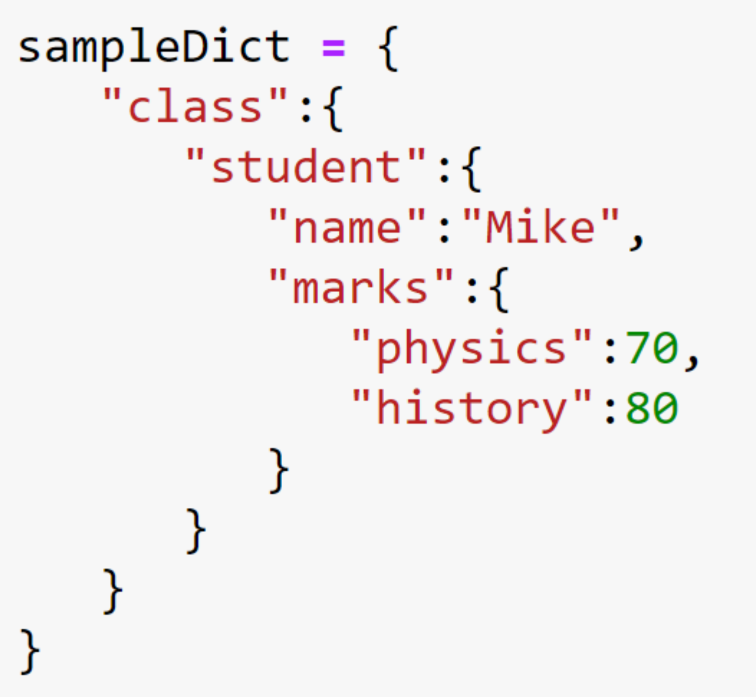
```
{'a': {'a': 1, 'b': 2}, 'b': {'a': 1, 'b': 2}}
```

How to access the value of a “history” subject

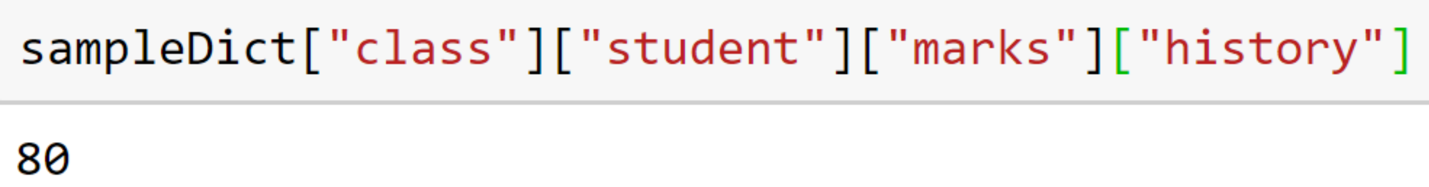


```
sampleDict = {
    "class":{
        "student":{
            "name":"Mike",
            "marks":{
                "physics":70,
                "history":80
            }
        }
    }
}
```

How to access the value of a “history” subject



```
sampleDict = {
    "class":{
        "student":{
            "name":"Mike",
            "marks":{
                "physics":70,
                "history":80
            }
        }
    }
}
```



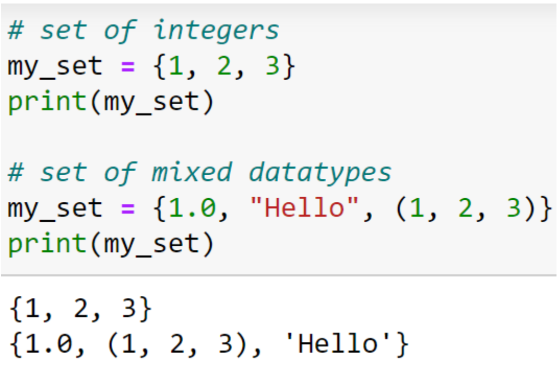
```
sampleDict["class"]["student"]["marks"]["history"]
```

```
80
```

Another data type - Set

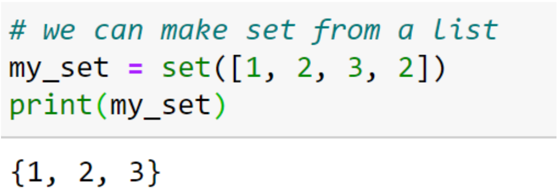
A set is a collection which is both **unordered** and **unindexed**.

- Sets are written with curly brackets.
- Every set element is **unique** (no duplicates) and must be **immutable** (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it.



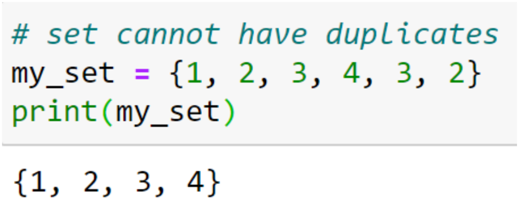
```
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```



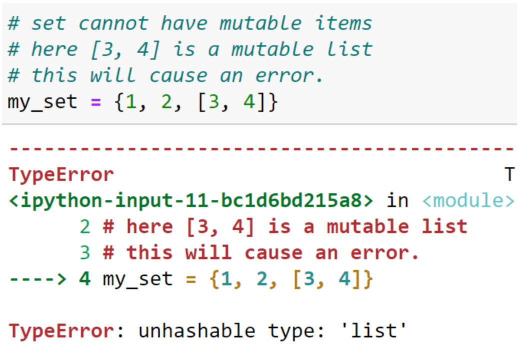
```
# we can make set from a list
my_set = set([1, 2, 3, 2])
print(my_set)
```

```
{1, 2, 3}
```



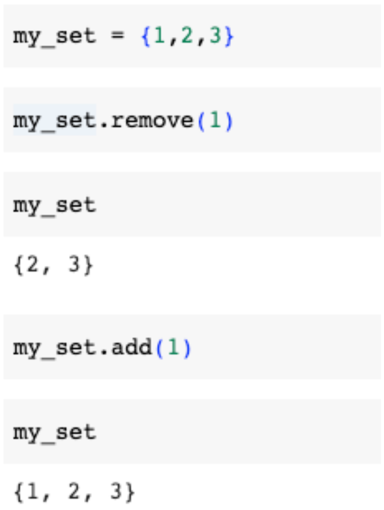
```
# set cannot have duplicates
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)
```

```
{1, 2, 3, 4}
```



```
# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.
my_set = {1, 2, [3, 4]}
```

```
---------------------------------------------------------------------------
TypeError                                 T
<ipython-input-11-bc1d6bd215a8> in <module>
      2 # here [3, 4] is a mutable list
      3 # this will cause an error.
----> 4 my_set = {1, 2, [3, 4]}

TypeError: unhashable type: 'list'
```



```
my_set = {1,2,3}
```

```
my_set.remove(1)
```

```
my_set
```

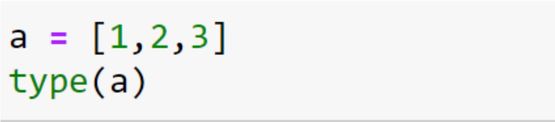
```
{2, 3}
```

```
my_set.add(1)
```

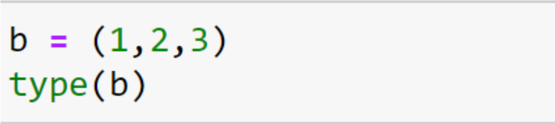
```
my_set
```

```
{1, 2, 3}
```

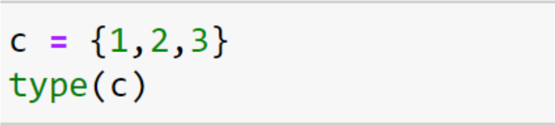
Data types



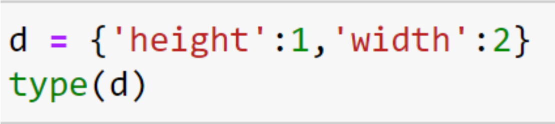
```
a = [1,2,3]
type(a)
```



```
b = (1,2,3)
type(b)
```

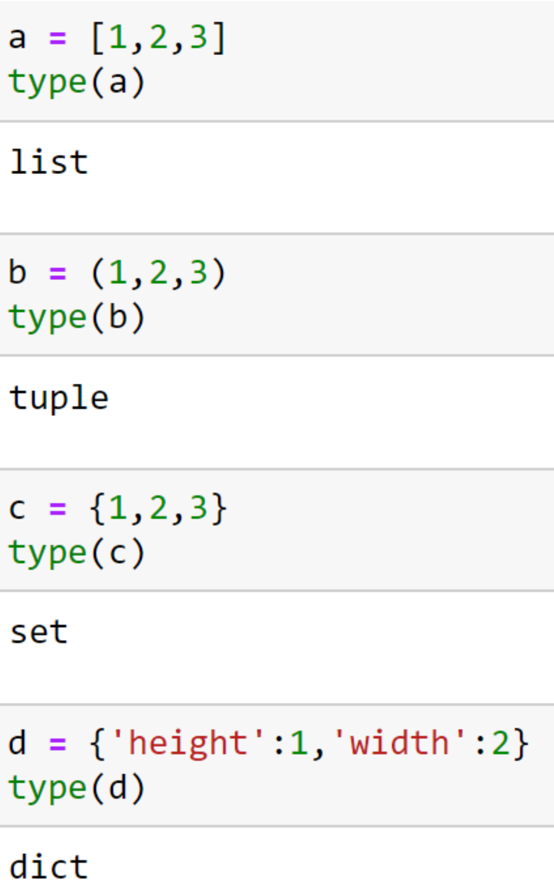


```
c = {1,2,3}
type(c)
```



```
d = {'height':1,'width':2}
type(d)
```

Data types



```
a = [1,2,3]
type(a)
```

```
list
```

```
b = (1,2,3)
type(b)
```

```
tuple
```

```
c = {1,2,3}
type(c)
```

```
set
```

```
d = {'height':1,'width':2}
type(d)
```

```
dict
```

Difference?

- Brackets
- Mutable
- Index
- Duplicates

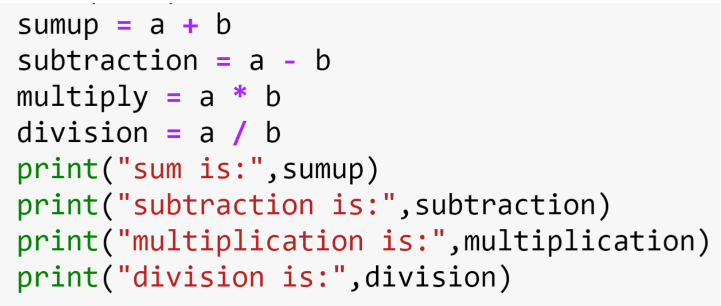
Lecture 7 Functions and Parameters

IDS 400

Programming for Data Science in Business

Reusable codes

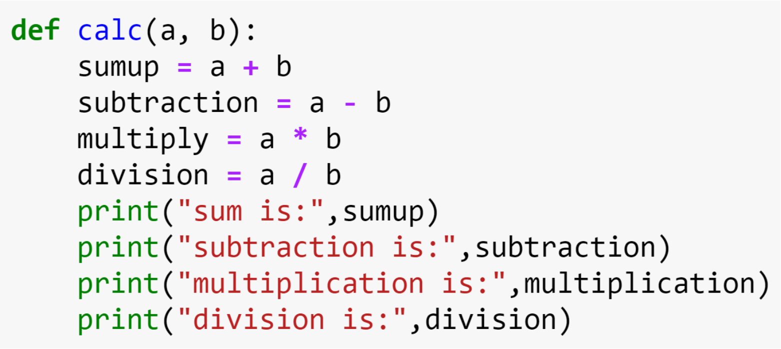
- Do the following operations for two numbers and print results: +, -, *, /



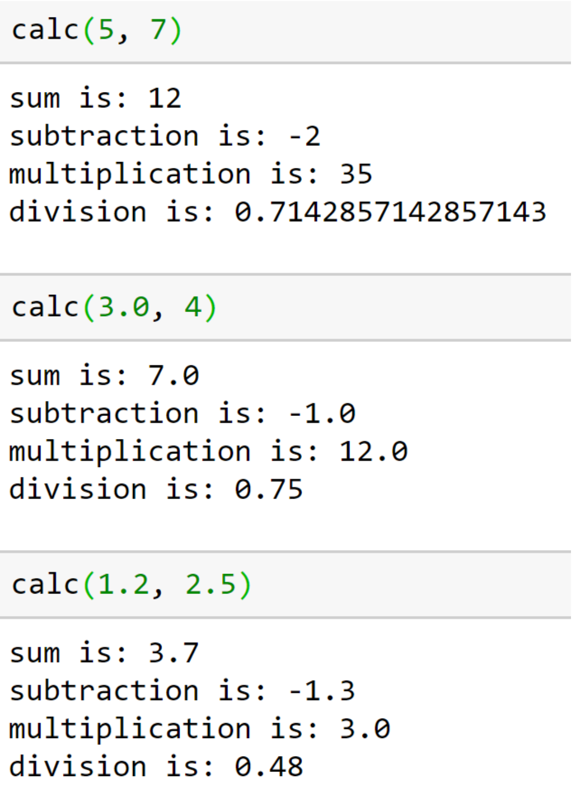
```
sumup = a + b
subtraction = a - b
multiply = a * b
division = a / b
print("sum is:",sumup)
print("subtraction is:",subtraction)
print("multiplication is:",multiplication)
print("division is:",division)
```

Reusable codes

- Do the following operations for two numbers and print results: +, -, *, /



```
def calc(a, b):
    sumup = a + b
    subtraction = a - b
    multiply = a * b
    division = a / b
    print("sum is:",sumup)
    print("subtraction is:",subtraction)
    print("multiplication is:",multiplication)
    print("division is:",division)
```



```
calc(5, 7)
```

```
sum is: 12
subtraction is: -2
multiplication is: 35
division is: 0.7142857142857143
```

```
calc(3.0, 4)
```

```
sum is: 7.0
subtraction is: -1.0
multiplication is: 12.0
division is: 0.75
```

```
calc(1.2, 2.5)
```

```
sum is: 3.7
subtraction is: -1.3
multiplication is: 3.0
division is: 0.48
```

Function

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

Python functions

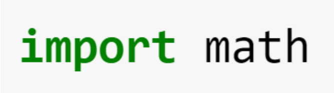
- There are two kinds of functions in Python.
  - Build-in functions that are provided as part of Python.

    ```
    input(), type(), float(),int(), …
    ```

  - Function that we define ourselves and use it.

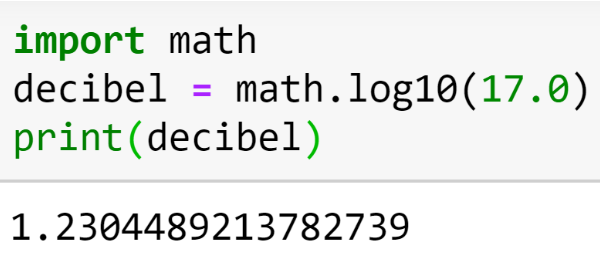
Build-in functions

- Math functions
  - ○ Python has a math module that provides most of the familiar mathematical functions.
  - ○ Before using all these functions, we have to import them:



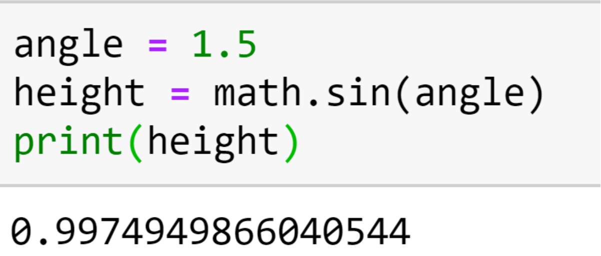
```
import math
```

  - ○ To call one of the functions, we have to *specify the name of the module and the name of the function*, separated by a dot. This format is also called **dot notation**.



```
import math
decibel = math.log10(17.0)
print(decibel)
```

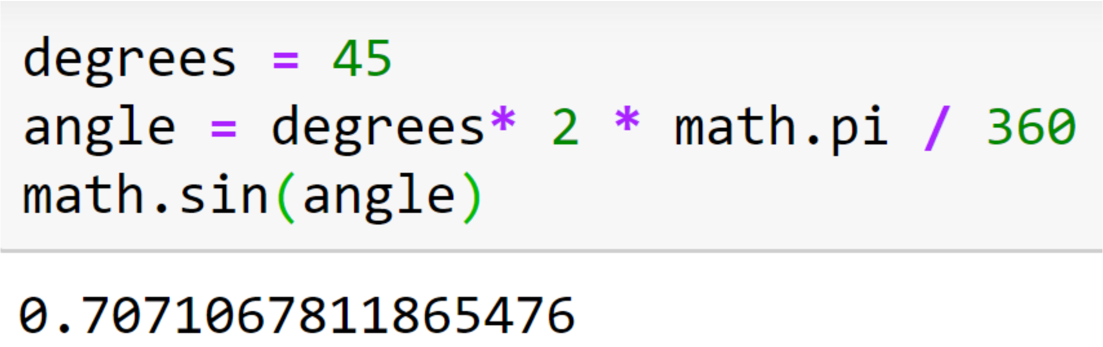
```
1.2304489213782739
```



```
angle = 1.5
height = math.sin(angle)
print(height)
```

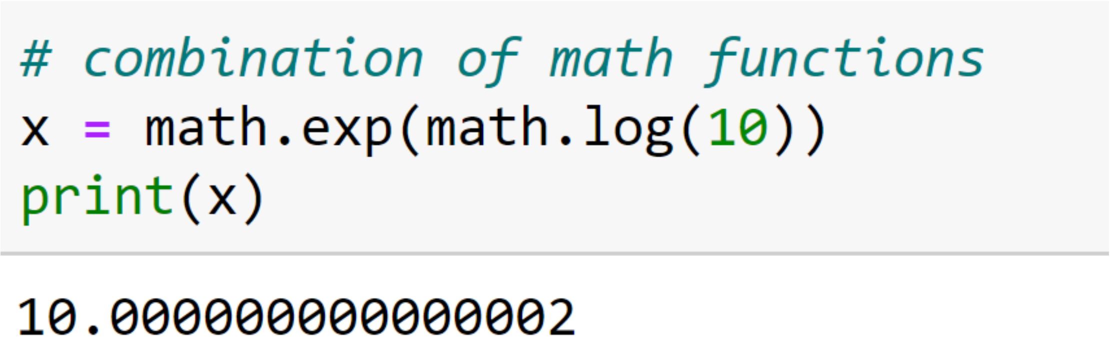
```
0.9974949866040544
```

Math function examples



```
degrees = 45
angle = degrees* 2 * math.pi / 360
math.sin(angle)
```

```
0.7071067811865476
```



```
# combination of math functions
x = math.exp(math.log(10))
print(x)
```

```
10.000000000000002
```

Many other math functions

Function name	Description
abs(**value**)	absolute value
ceil(**value**)	rounds up
cos(**value**)	cosine, in radians
degrees(**value**)	convert radians to degrees
floor(**value**)	rounds down
log(**value**, **base**)	logarithm in any base
log10(**value**)	logarithm, base 10
max(**value1**, **value2, ...**)	larger of two (or more) values
min(**value1**, **value2, ...**)	smaller of two (or more) values
radians(**value**)	convert degrees to radians
round(**value**)	nearest whole number
sin(**value**)	sine, in radians
sqrt(**value**)	square root
tan(**value**)	tangent

Constant	Description
e	2.7182818...
pi	3.1415926...

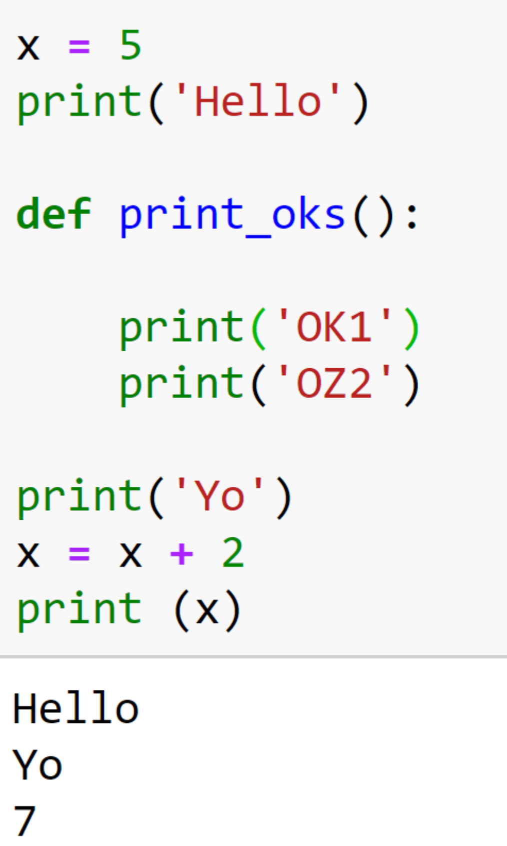
Defining your own functions

- Simple rules to define a function
  - Function blocks begin with the keyword ***def***
  - Followed by the *function name*, *parentheses ()*, and *a colon :*
  - A **parameter** is a variable which we use in the function definition.
  - Any parameters should be placed within these parentheses.
  - The code block within every function is indented.

```
def FunctionName (parameters):
    <statement> in the function body
    ...
<statement> outside the function body
```

Building your own functions

- Defining a function doesn’t mean we execute the body of the function.

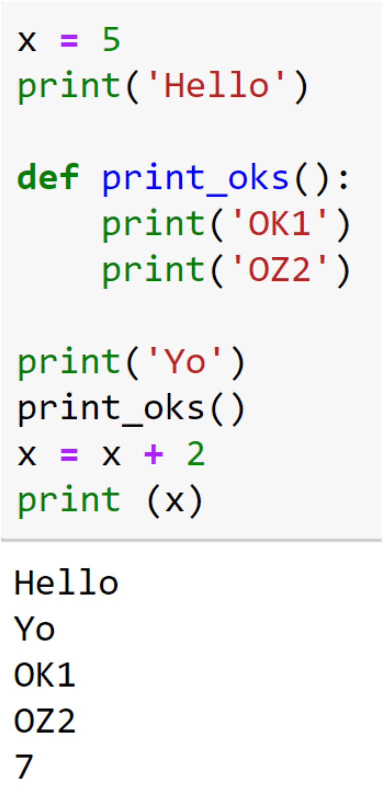


```python
x = 5
print('Hello')

def print_oks():

    print('OK1')
    print('OZ2')

print('Yo')
x = x + 2
print (x)
```

```
Hello
Yo
7
```

Calling a function

- Once we have defined a function, we can call (or invoke) it as many times as we like.
- This is a store and reuse pattern.



```
x = 5
print('Hello')

def print_oks():
    print('OK1')
    print('OZ2')

print('Yo')
print_oks()
x = x + 2
print (x)
```

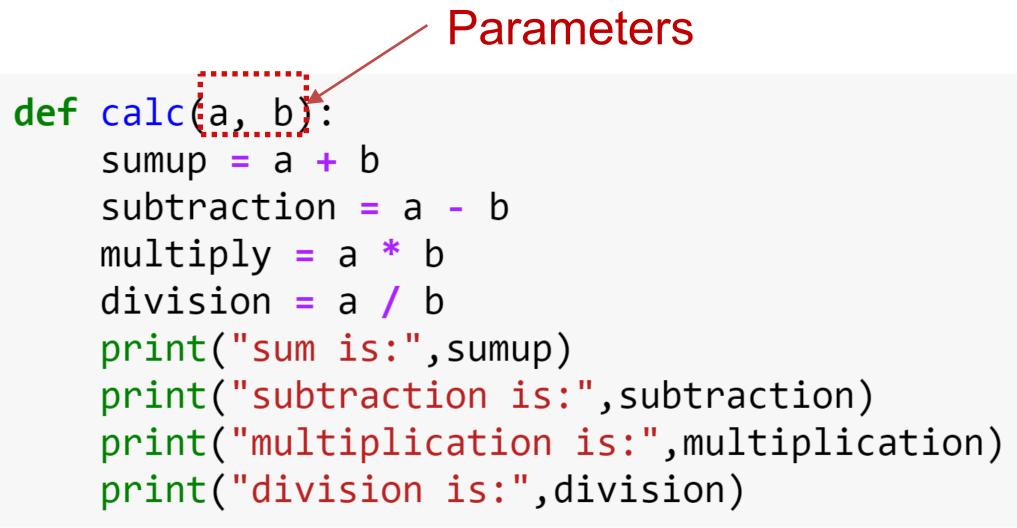
```
Hello
Yo
OK1
OZ2
7
```

Arguments & Parameters

- An **argument** is a value we pass into the function as its input when we call the function.
- Sometimes, the function doesn't need input (**no arguments**).
- We use arguments so we can direct the function to do different kinds of work when we call it at different times.
- We put the arguments in *parentheses after the name of the function*.

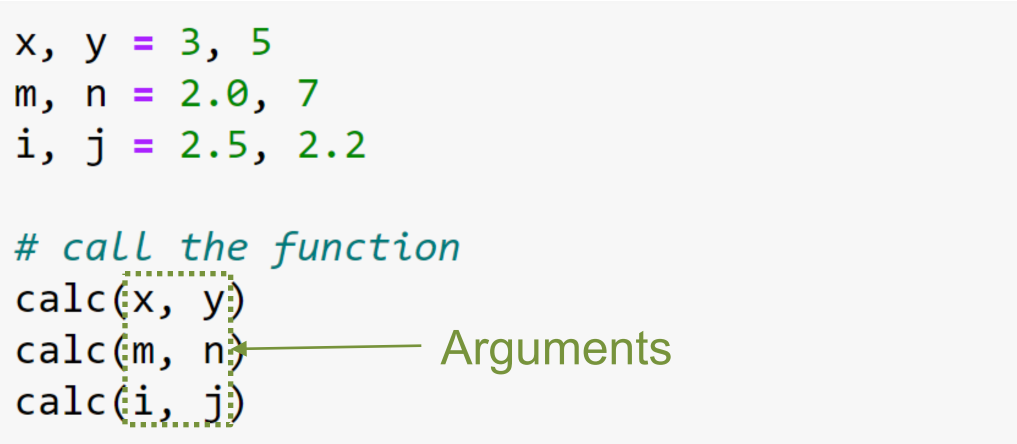
- A **parameter** is a variable which we use in the function definition.
- It is a "**handle**" that allows the code in the function to access the arguments for a particular function invocation.

Arguments & Parameters



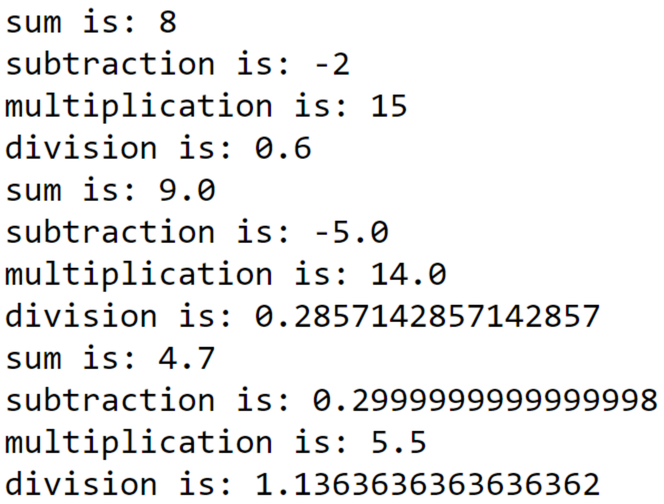
Parameters

```
def calc(a, b):
    sumup = a + b
    subtraction = a - b
    multiply = a * b
    division = a / b
    print("sum is:",sumup)
    print("subtraction is:",subtraction)
    print("multiplication is:",multiplication)
    print("division is:",division)
```



```
x, y = 3, 5
m, n = 2.0, 7
i, j = 2.5, 2.2

# call the function
calc(x, y)
calc(m, n)
calc(i, j)
```

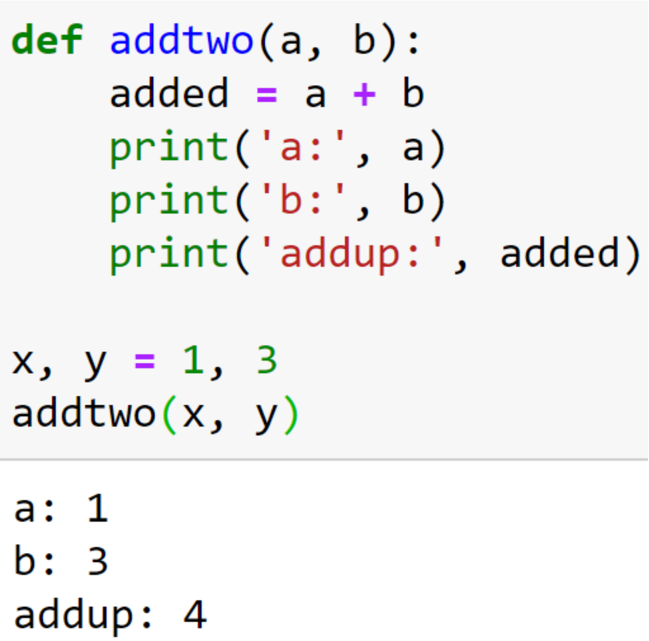
Arguments



```
sum is: 8
subtraction is: -2
multiplication is: 15
division is: 0.6
sum is: 9.0
subtraction is: -5.0
multiplication is: 14.0
division is: 0.2857142857142857
sum is: 4.7
subtraction is: 0.2999999999999998
multiplication is: 5.5
division is: 1.1363636363636362
```

Multiple parameters/ argument

- We can define more than one parameter in the function definition.
- We simply add more arguments when we call the function.
- We match the number and the order of arguments and parameters.



```python
def addtwo(a, b):
    added = a + b
    print('a:', a)
    print('b:', b)
    print('addup:', added)

x, y = 1, 3
addtwo(x, y)
```

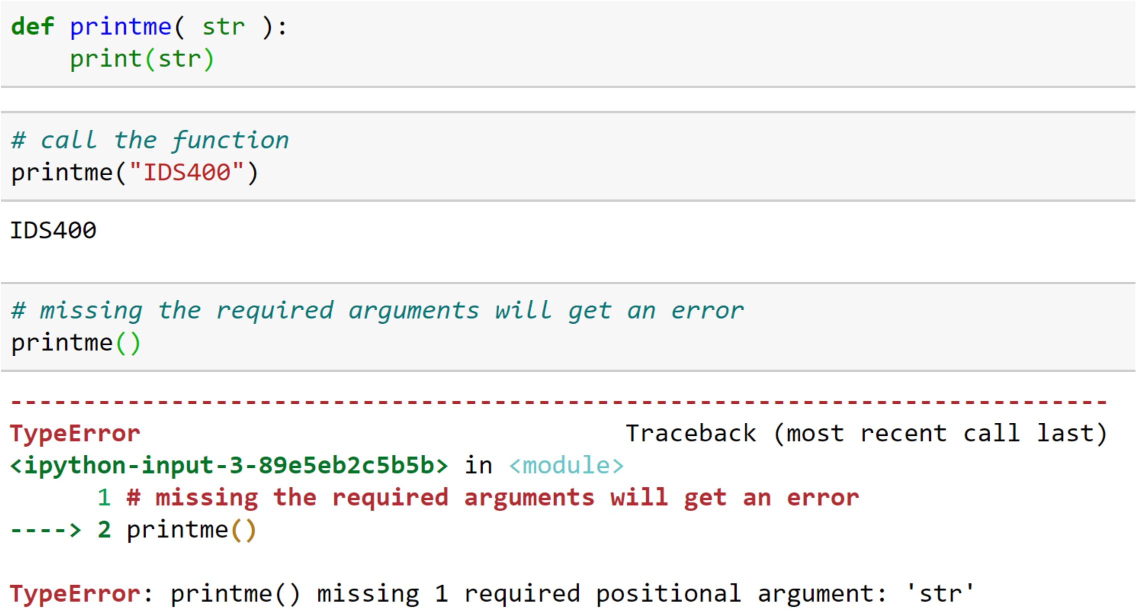
```
a: 1
b: 3
addup: 4
```

Arguments

- In Python, there are four types of arguments.
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function correct positional order.
- The number of arguments in the function call should **match exactly** with the parameters in the function definition.



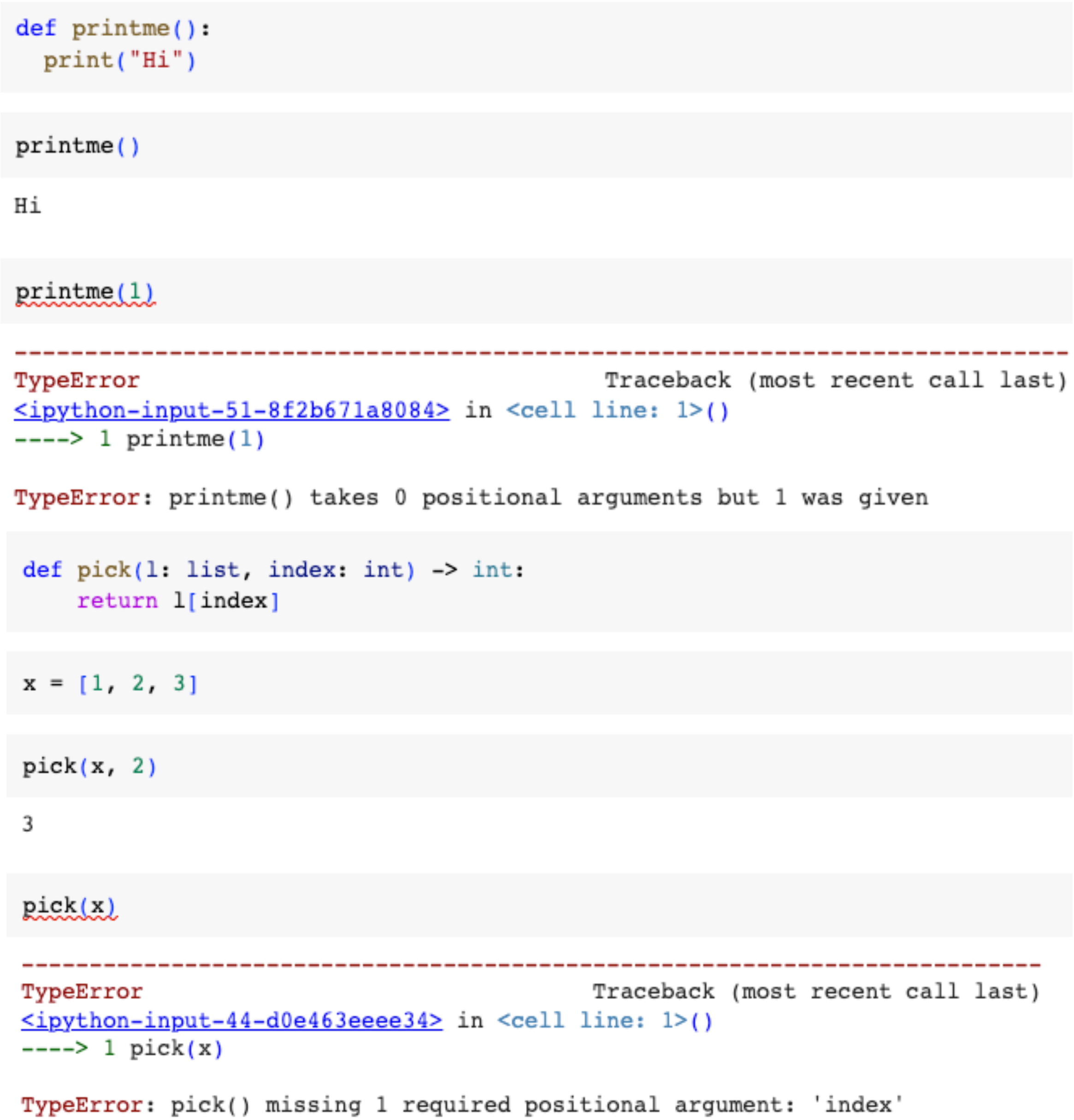
```
def printme( str ):
    print(str)
```

```
# call the function
printme("IDS400")
```

```
IDS400
```

```
# missing the required arguments will get an error
printme()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-89e5eb2c5b5b> in <module>
      1 # missing the required arguments will get an error
----> 2 printme()

TypeError: printme() missing 1 required positional argument: 'str'
```



```
def printme():
  print("Hi")
```

```
printme()
```

```
Hi
```

```
printme(1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-8f2b671a8084> in <cell line: 1>()
----> 1 printme(1)

TypeError: printme() takes 0 positional arguments but 1 was given
```

```
def pick(l: list, index: int) -> int:
    return l[index]
```

```
x = [1, 2, 3]
```

```
pick(x, 2)
```

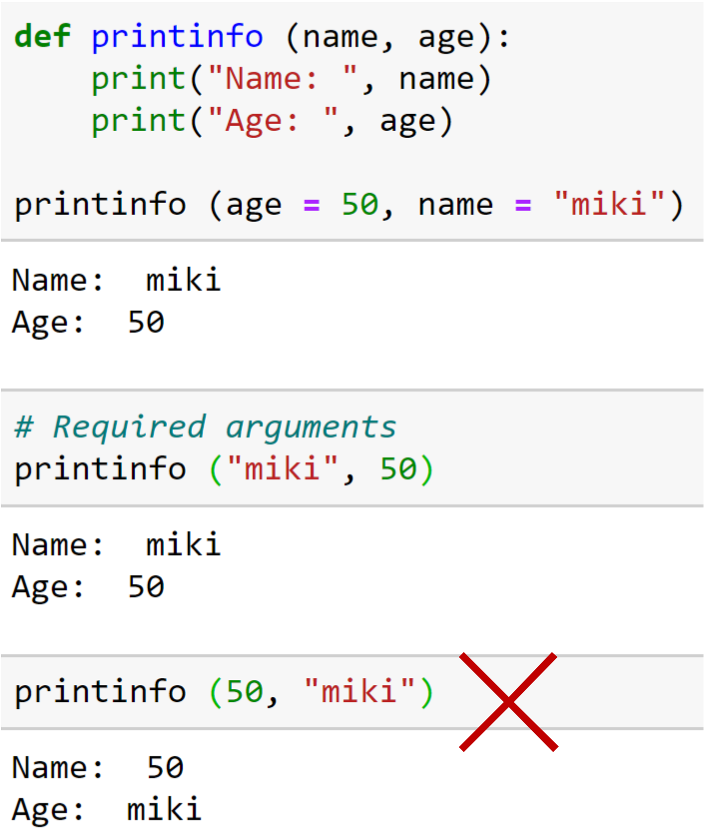
```
3
```

```
pick(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-d0e463eeee34> in <cell line: 1>()
----> 1 pick(x)

TypeError: pick() missing 1 required positional argument: 'index'
```

Keyword arguments

- Keyword arguments are related to the function calls.
- When you use keyword arguments in a function call, *the caller identifies the arguments by the parameter name*.
- This allows you to *place them out of order* because the Python interpreter is able to use the keywords provided to match the values with parameters.



```
def printinfo (name, age):
    print("Name: ", name)
    print("Age: ", age)

printinfo (age = 50, name = "miki")
```

```
Name:  miki
Age:  50
```

```
# Required arguments
printinfo ("miki", 50)
```

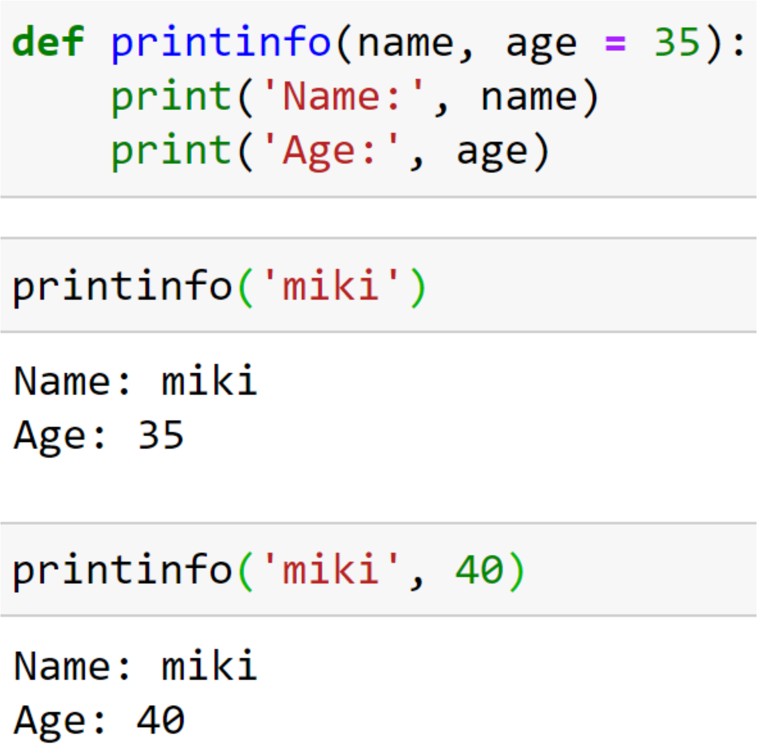
```
Name:  miki
Age:  50
```

```
printinfo (50, "miki")
```

```
Name:  50
Age:  miki
```

Default argument

- A default argument is an argument that assumes *a default value if a value is not provided* in the function call for that argument.



```
def printinfo(name, age = 35):
    print('Name:', name)
    print('Age:', age)
```

```
printinfo('miki')
```

```
Name: miki
Age: 35
```

```
printinfo('miki', 40)
```

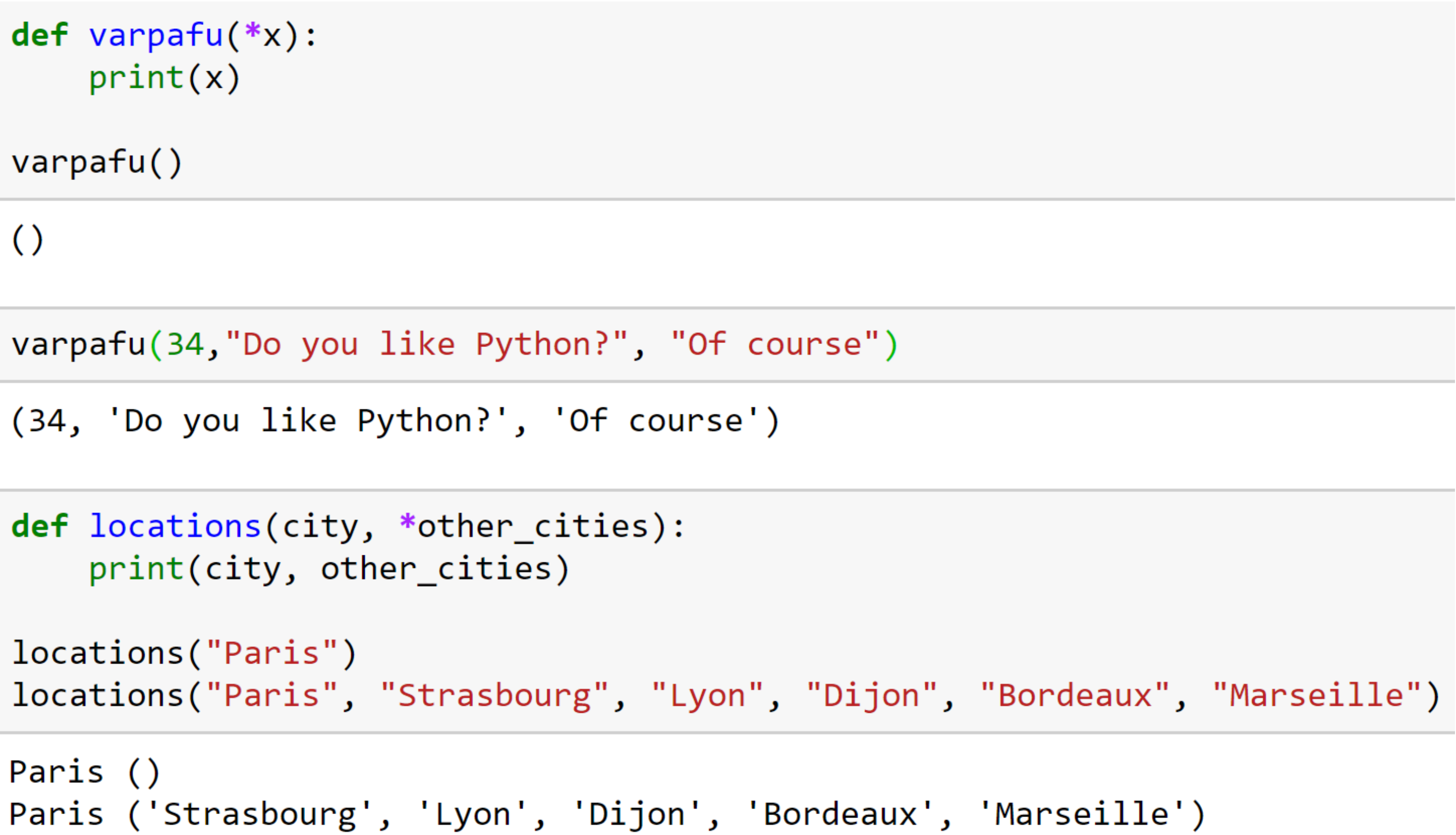
```
Name: miki
Age: 40
```

Variable-length arguments

- You may need to process a function for *more arguments than you specified* while defining the function.
- Variable-length arguments are not named in the function definition.
- Use asterisk* **before the variable name** that holds the values of all other variable arguments in the definition.
- This variable remains empty if no additional arguments are specified during the call.

```
def FunctionName (<formal args>, *var_args_tuple):
    <statement> in the function body
    ...
<statement> outside the function body
```

Examples



```
def varpafu(*x):
    print(x)

varpafu()
```

```
()
```

```
varpafu(34,"Do you like Python?", "Of course")
```

```
(34, 'Do you like Python?', 'Of course')
```

```
def locations(city, *other_cities):
    print(city, other_cities)

locations("Paris")
locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux", "Marseille")
```

```
Paris ()
Paris ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseille')
```

Anonymous functions

- Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called ***lambda***.
- Anonymous functions are NOT declared in the standard manner by using the **def** keyword.
- Instead, we use the **lambda** keyword to create small anonymous functions.

```
Lambda: expression
Lambda arg1, arg2, …, argN: expression
```

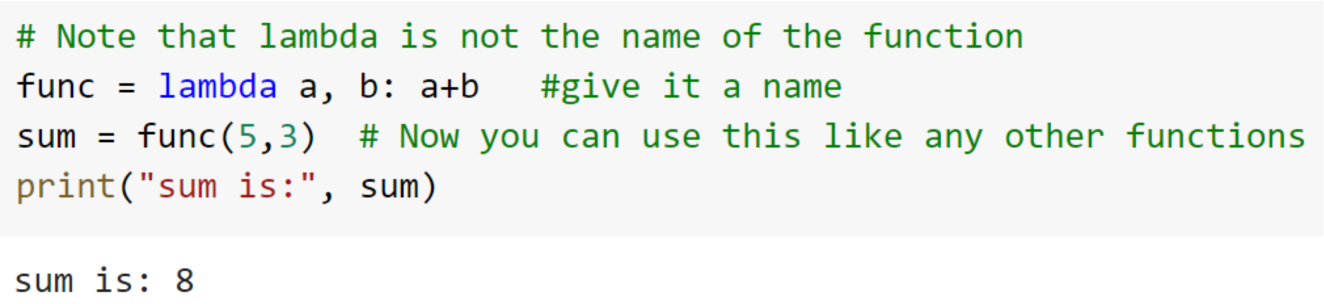
Anonymous functions

- Function objects returned by running lambda expressions work exactly the same as those created and assigned by ***def***s. However, there are a few differences that make lambda useful in specialized roles:
  - lambda is an **expression**, not a statement.
  - lambda's body is **a single expression**, not a block of statements or multiple expressions.
  - lambda is designed for coding simple functions, and def handles larger tasks.

```
Lambda arg1, arg2, …, argN: expression
```

Anonymous function example

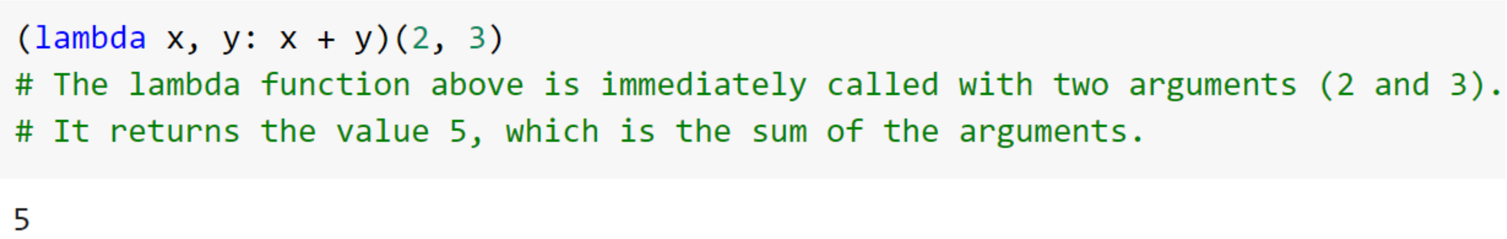
- We can use lambda expression by giving it a name.



```
# Note that lambda is not the name of the function
func = lambda a, b: a+b   #give it a name
sum = func(5,3)  # Now you can use this like any other functions
print("sum is:", sum)
```

```
sum is: 8
```

- We can also execute it immediately.

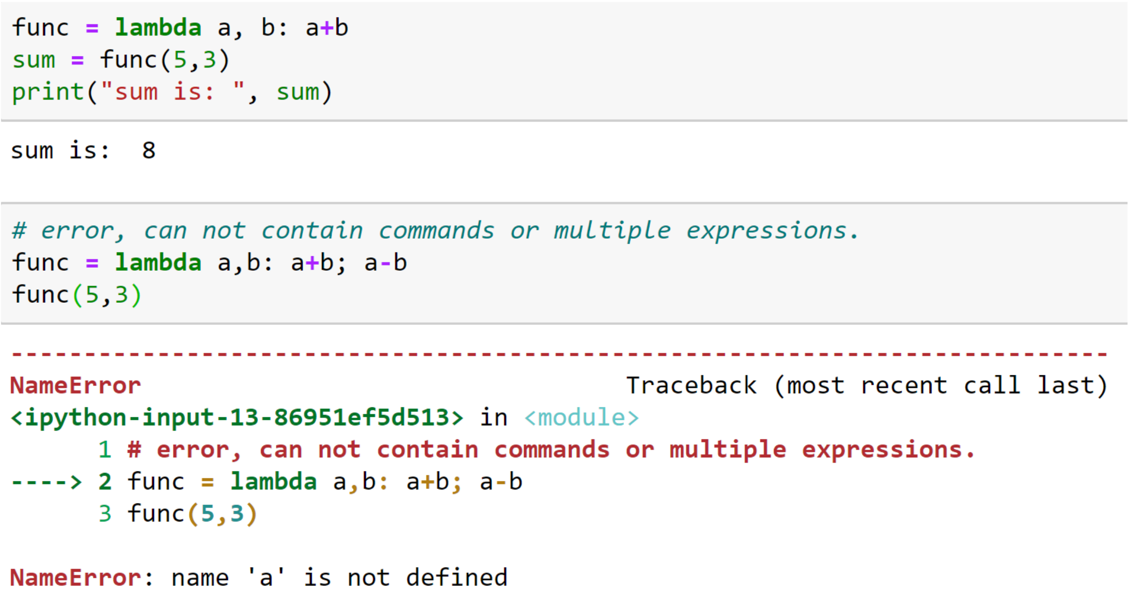


```
(lambda x, y: x + y)(2, 3)
# The lambda function above is immediately called with two arguments (2 and 3).
# It returns the value 5, which is the sum of the arguments.
```

```
5
```

Anonymous function

- Lambda forms can *take any number of arguments* but *return just one value* in the form of an expression. They can NOT contain **statements or multiple expressions**.



```
func = lambda a, b: a+b
sum = func(5,3)
print("sum is: ", sum)
```

```
sum is:  8
```

```
# error, can not contain commands or multiple expressions.
func = lambda a,b: a+b; a-b
func(5,3)
```

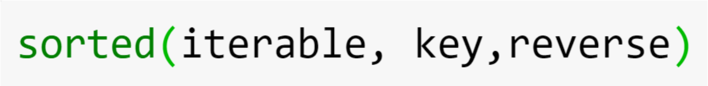
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-86951ef5d513> in <module>
      1 # error, can not contain commands or multiple expressions.
----> 2 func = lambda a,b: a+b; a-b
      3 func(5,3)

NameError: name 'a' is not defined
```

Anonymous function example

- Lambda expressions are quite useful when you need a short, throwaway function, especially when you only use them once.
- Common applications are sorting and filtering data.

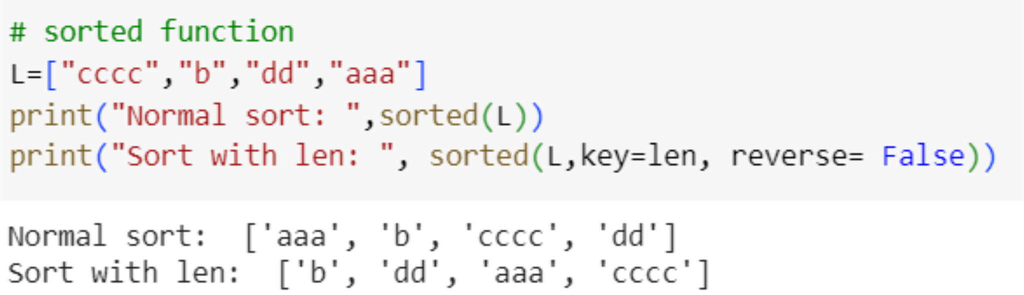
Anonymous function example

- Sorted function:



```
sorted(iterable, key,reverse)
```

  - iterable can be a list, tuple or dictionary.
  - key is a function that serve as a key or basis of sort comparison. The function corresponding to key will apply the function on each element of this iterable item and return a value. This value is then used for sorting.
  - reverse is set to true if sort in descending order (from the largest value to the smallest value).

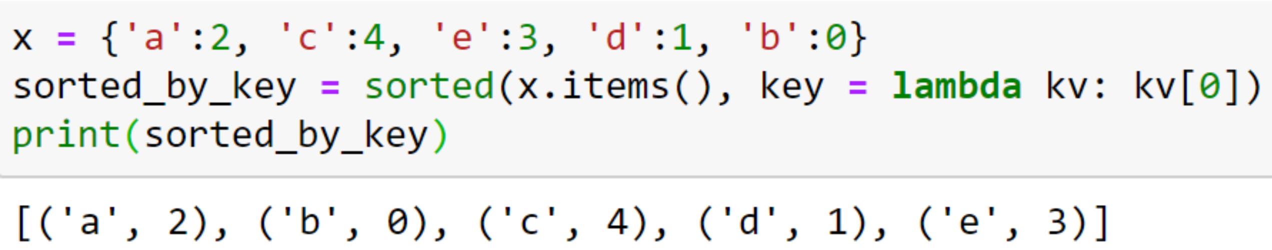


```
# sorted function
L=["cccc","b","dd","aaa"]
print("Normal sort: ",sorted(L))
print("Sort with len: ", sorted(L,key=len, reverse= False))
```

```
Normal sort:  ['aaa', 'b', 'cccc', 'dd']
Sort with len:  ['b', 'dd', 'aaa', 'cccc']
```

Anonymous function example

- *Task 1*: Sort a dictionary by key.



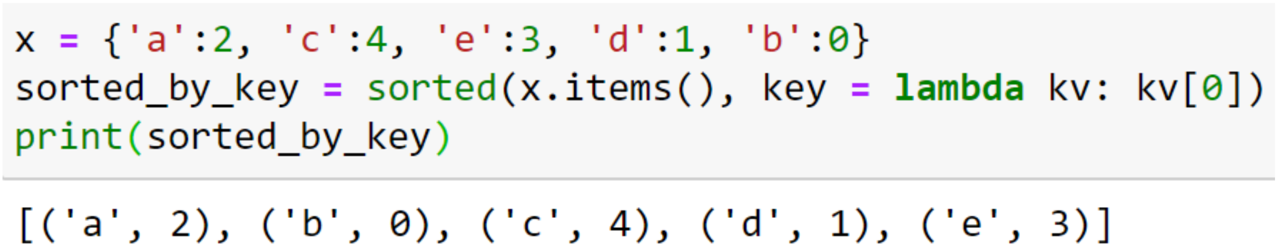
```
x = {'a':2, 'c':4, 'e':3, 'd':1, 'b':0}
sorted_by_key = sorted(x.items(), key = lambda kv: kv[0])
print(sorted_by_key)
```

```
[('a', 2), ('b', 0), ('c', 4), ('d', 1), ('e', 3)]
```

- *Question:* How to sort a dictionary by value?

Anonymous function example

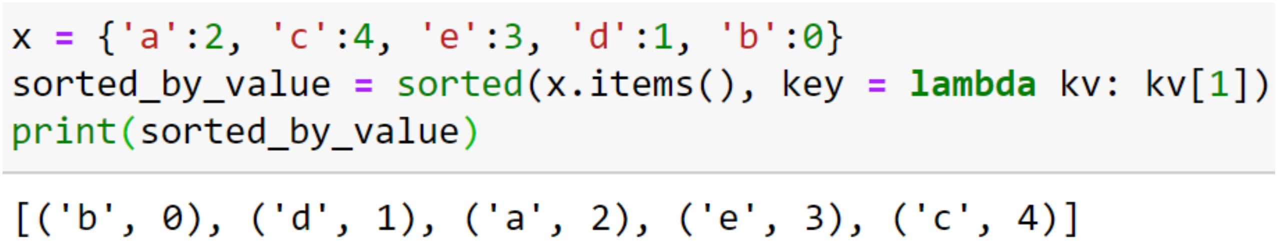
- *Task 1*: Sort a dictionary by key.



```
x = {'a':2, 'c':4, 'e':3, 'd':1, 'b':0}
sorted_by_key = sorted(x.items(), key = lambda kv: kv[0])
print(sorted_by_key)
```

```
[('a', 2), ('b', 0), ('c', 4), ('d', 1), ('e', 3)]
```

- *Question:* How to sort a dictionary by value?

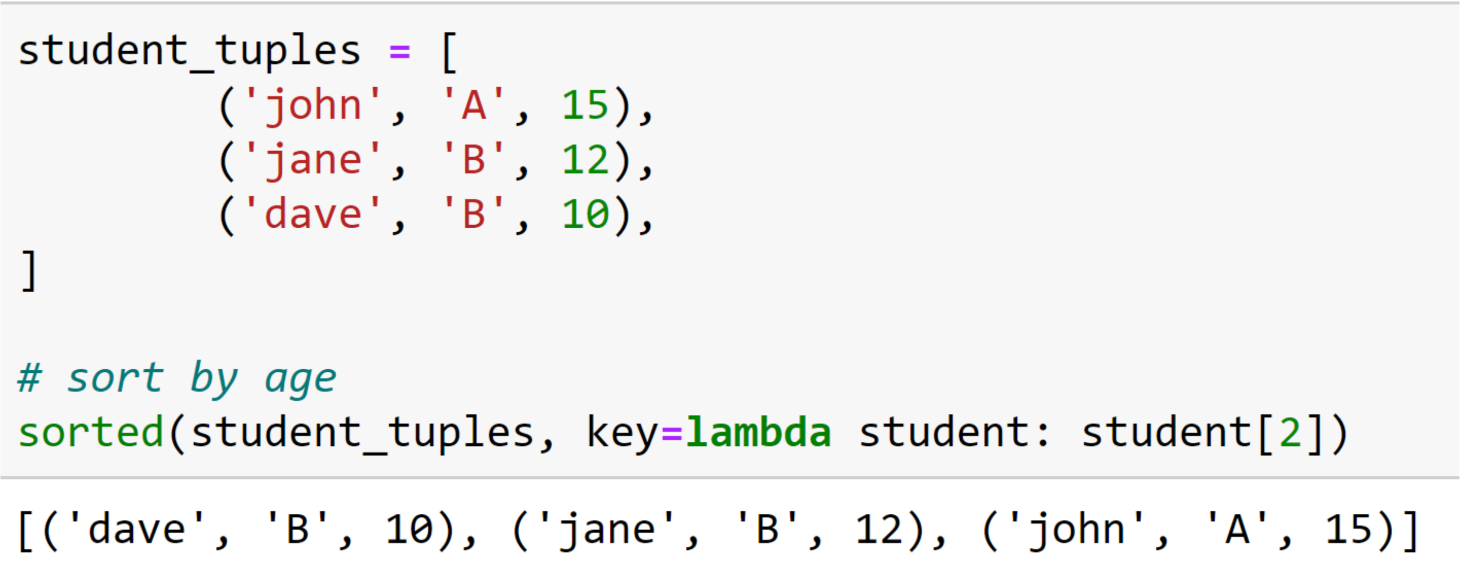


```
x = {'a':2, 'c':4, 'e':3, 'd':1, 'b':0}
sorted_by_value = sorted(x.items(), key = lambda kv: kv[1])
print(sorted_by_value)
```

```
[('b', 0), ('d', 1), ('a', 2), ('e', 3), ('c', 4)]
```

Sort using anonymous function

- *Task 2*: Suppose that you have a list of tuples, and each element inside the tuples represents a student’s name, grade and age. We would like to sort it by age.

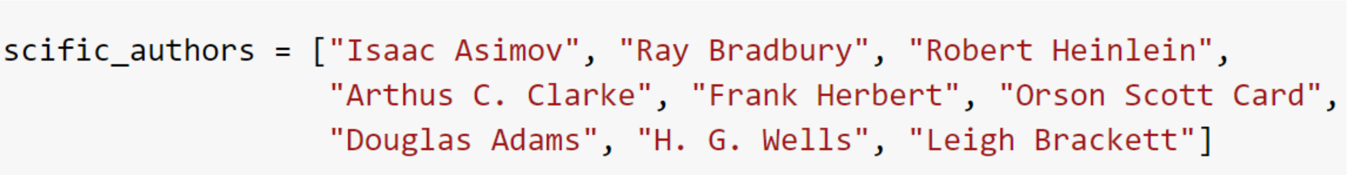


```python
student_tuples = [
        ('john', 'A', 15),
        ('jane', 'B', 12),
        ('dave', 'B', 10),
]

# sort by age
sorted(student_tuples, key=lambda student: student[2])
```

```
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Sort using anonymous function

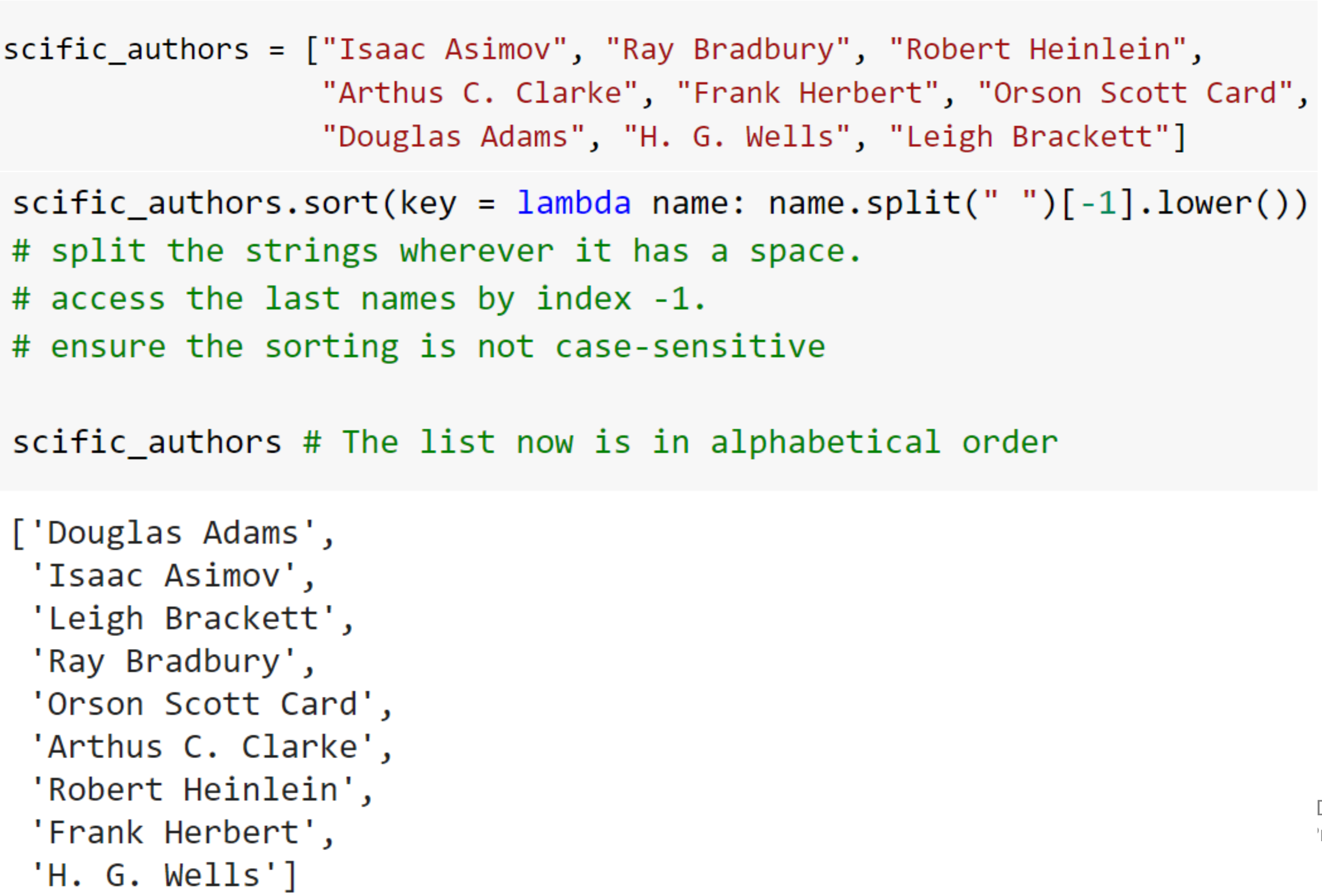
- *Task 3*: Suppose that you have a list of science fiction authors. We would like to sort this list by last name.



```
scific_authors = ["Isaac Asimov", "Ray Bradbury", "Robert Heinlein",
                  "Arthus C. Clarke", "Frank Herbert", "Orson Scott Card",
                  "Douglas Adams", "H. G. Wells", "Leigh Brackett"]
```

Sort using anonymous function

- *Task 3*: Suppose that you have a list of science fiction authors. We would like to sort this list by last name.



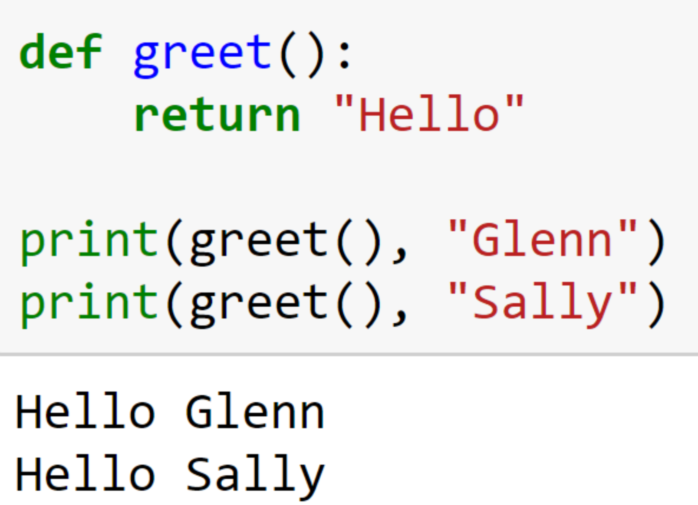
```
scific_authors = ["Isaac Asimov", "Ray Bradbury", "Robert Heinlein",
                  "Arthus C. Clarke", "Frank Herbert", "Orson Scott Card",
                  "Douglas Adams", "H. G. Wells", "Leigh Brackett"]
```

```
scific_authors.sort(key = lambda name: name.split(" ")[-1].lower())
# split the strings wherever it has a space.
# access the last names by index -1.
# ensure the sorting is not case-sensitive

scific_authors # The list now is in alphabetical order
```

```
['Douglas Adams',
 'Isaac Asimov',
 'Leigh Brackett',
 'Ray Bradbury',
 'Orson Scott Card',
 'Arthus C. Clarke',
 'Robert Heinlein',
 'Frank Herbert',
 'H. G. Wells']
```

The return statement

- Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression.
- The *return* keyword is used for this.
- A return statement with no arguments is the same as return none.

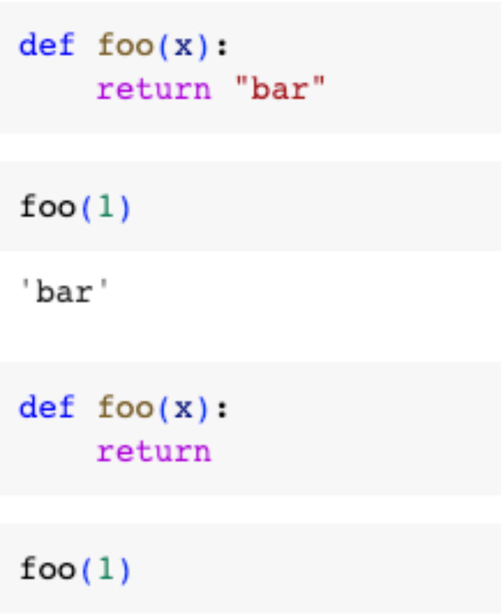


```
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

```
Hello Glenn
Hello Sally
```

Void (non-fruitful) functions

- When a function does not return a value, we call it a “void” function.
- Functions that return values are “fruitful” functions.
- Void functions are “not fruitful”.



```
def foo(x):
    return "bar"
```

```
foo(1)
```

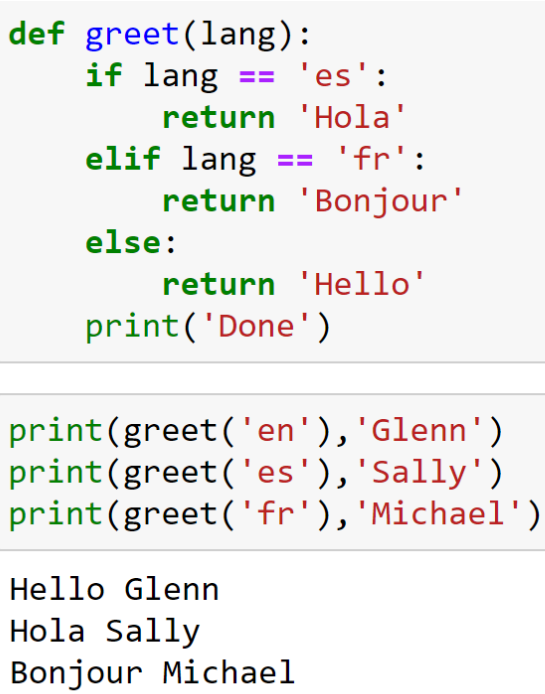
```
'bar'
```

```
def foo(x):
    return
```

```
foo(1)
```

The return statement

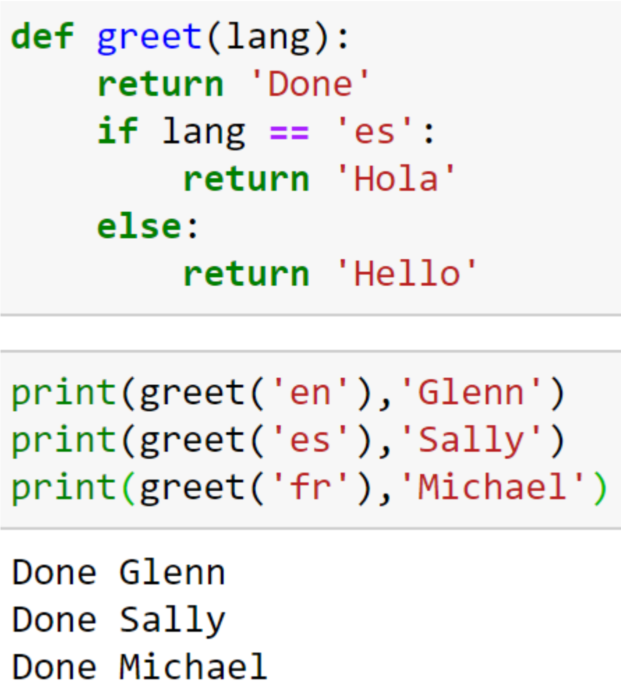
- The return statement *<u>ends the function execution </u>*and “sends back” the result of the function.



```
def greet(lang):
    if lang == 'es':
        return 'Hola'
    elif lang == 'fr':
        return 'Bonjour'
    else:
        return 'Hello'
    print('Done')
```

```
print(greet('en'),'Glenn')
print(greet('es'),'Sally')
print(greet('fr'),'Michael')
```

```
Hello Glenn
Hola Sally
Bonjour Michael
```



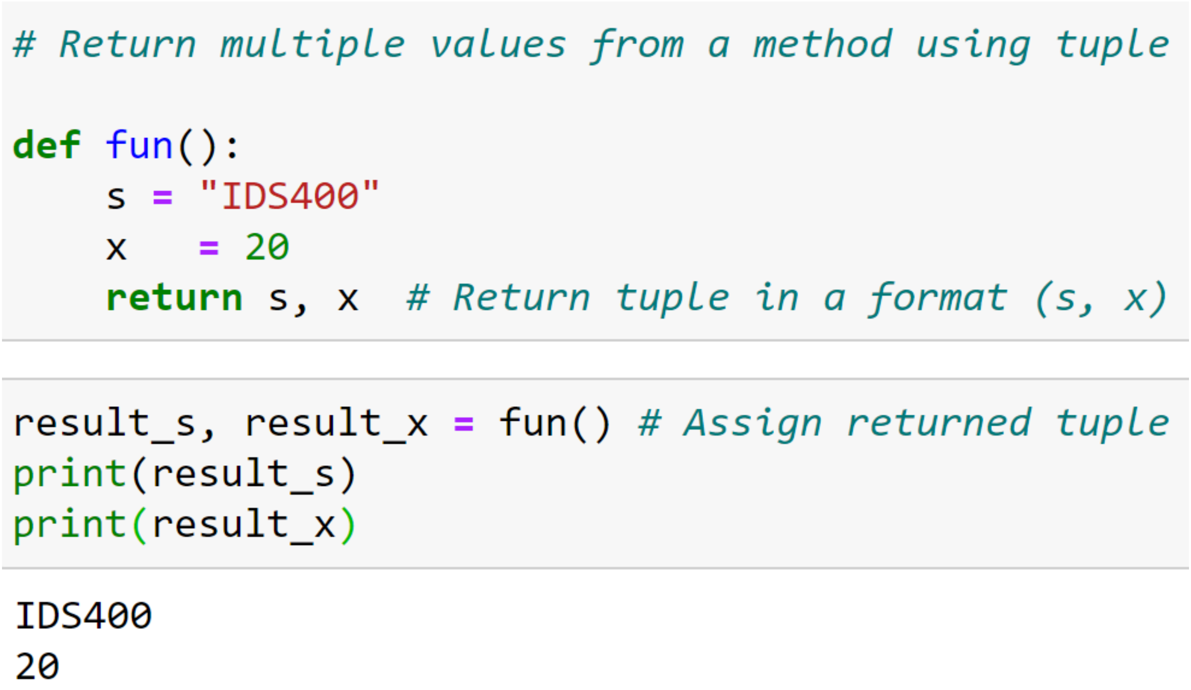
```
def greet(lang):
    return 'Done'
    if lang == 'es':
        return 'Hola'
    else:
        return 'Hello'
```

```
print(greet('en'),'Glenn')
print(greet('es'),'Sally')
print(greet('fr'),'Michael')
```

```
Done Glenn
Done Sally
Done Michael
```

The return statement

- We can also return multiple values.



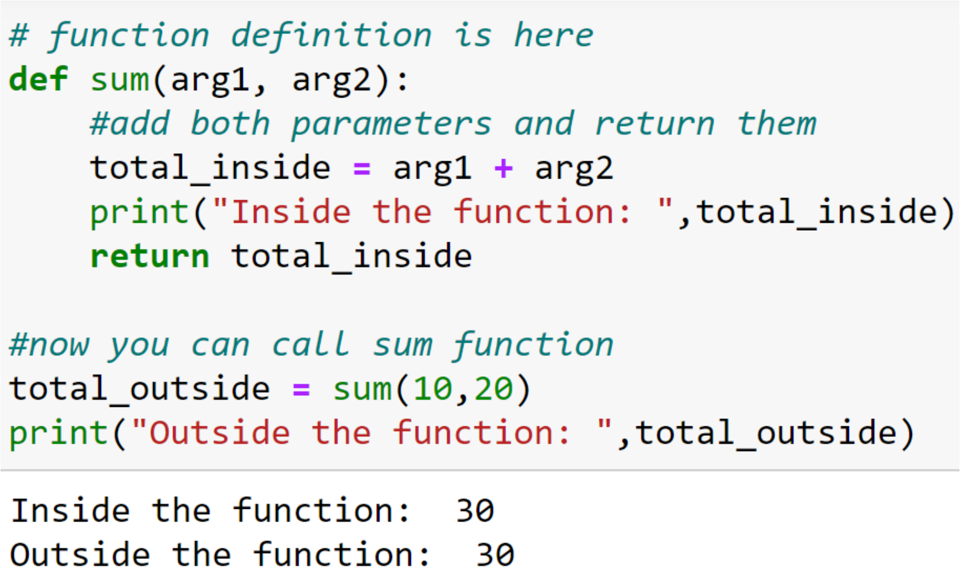
```python
# Return multiple values from a method using tuple

def fun():
    s = "IDS400"
    x   = 20
    return s, x  # Return tuple in a format (s, x)
```

```python
result_s, result_x = fun() # Assign returned tuple
print(result_s)
print(result_x)
```

```
IDS400
20
```

An example of variable scope

- A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

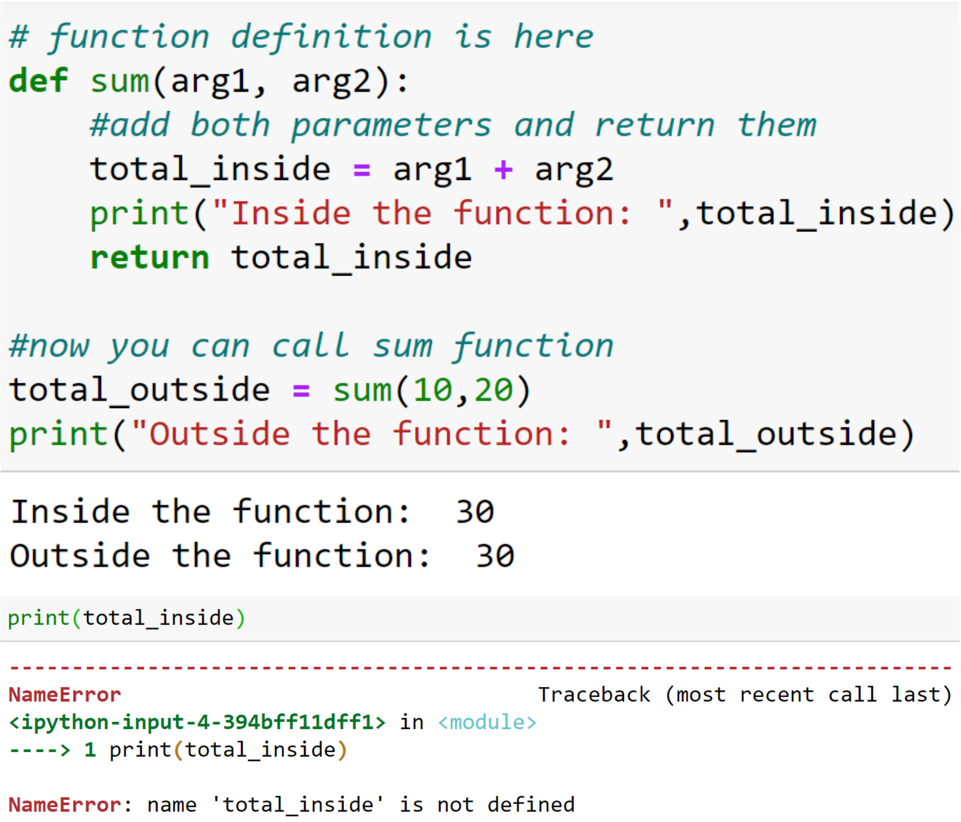


```python
# function definition is here
def sum(arg1, arg2):
    #add both parameters and return them
    total_inside = arg1 + arg2
    print("Inside the function: ",total_inside)
    return total_inside

#now you can call sum function
total_outside = sum(10,20)
print("Outside the function: ",total_outside)
```

```
Inside the function:  30
Outside the function:  30
```

An example of variable scope

- A variable created inside a function belongs to the local scope of that function and can only be used inside that function.



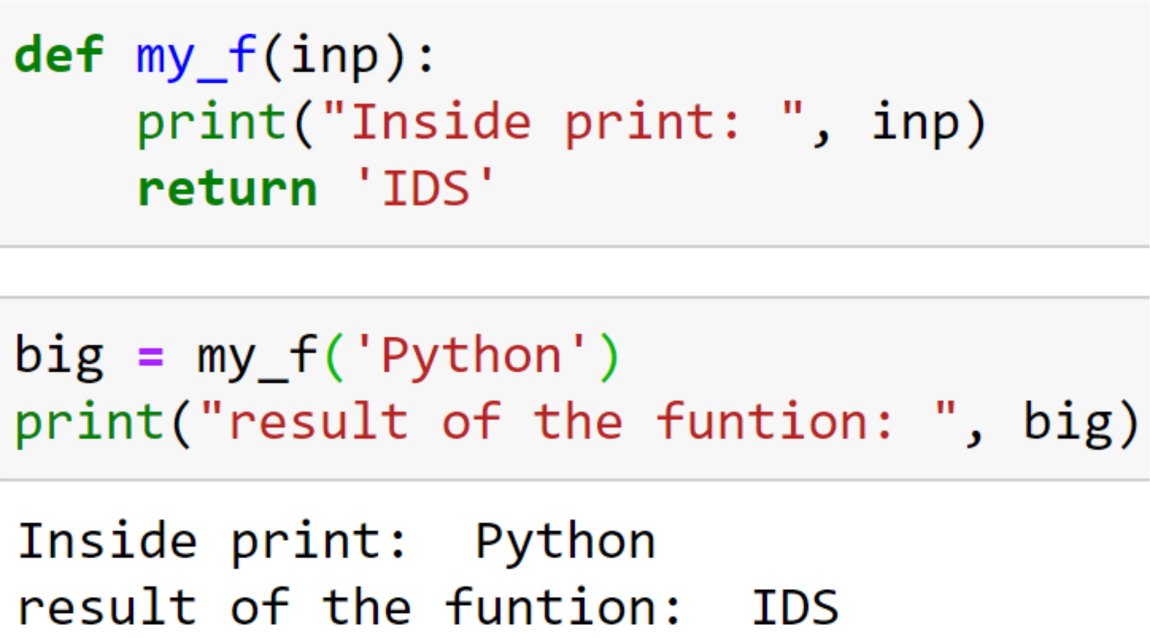
```python
# function definition is here
def sum(arg1, arg2):
    #add both parameters and return them
    total_inside = arg1 + arg2
    print("Inside the function: ",total_inside)
    return total_inside

#now you can call sum function
total_outside = sum(10,20)
print("Outside the function: ",total_outside)
```

```
Inside the function:  30
Outside the function:  30
```

```python
print(total_inside)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-394bff11dff1> in <module>
----> 1 print(total_inside)

NameError: name 'total_inside' is not defined
```

Arguments, parameters and results

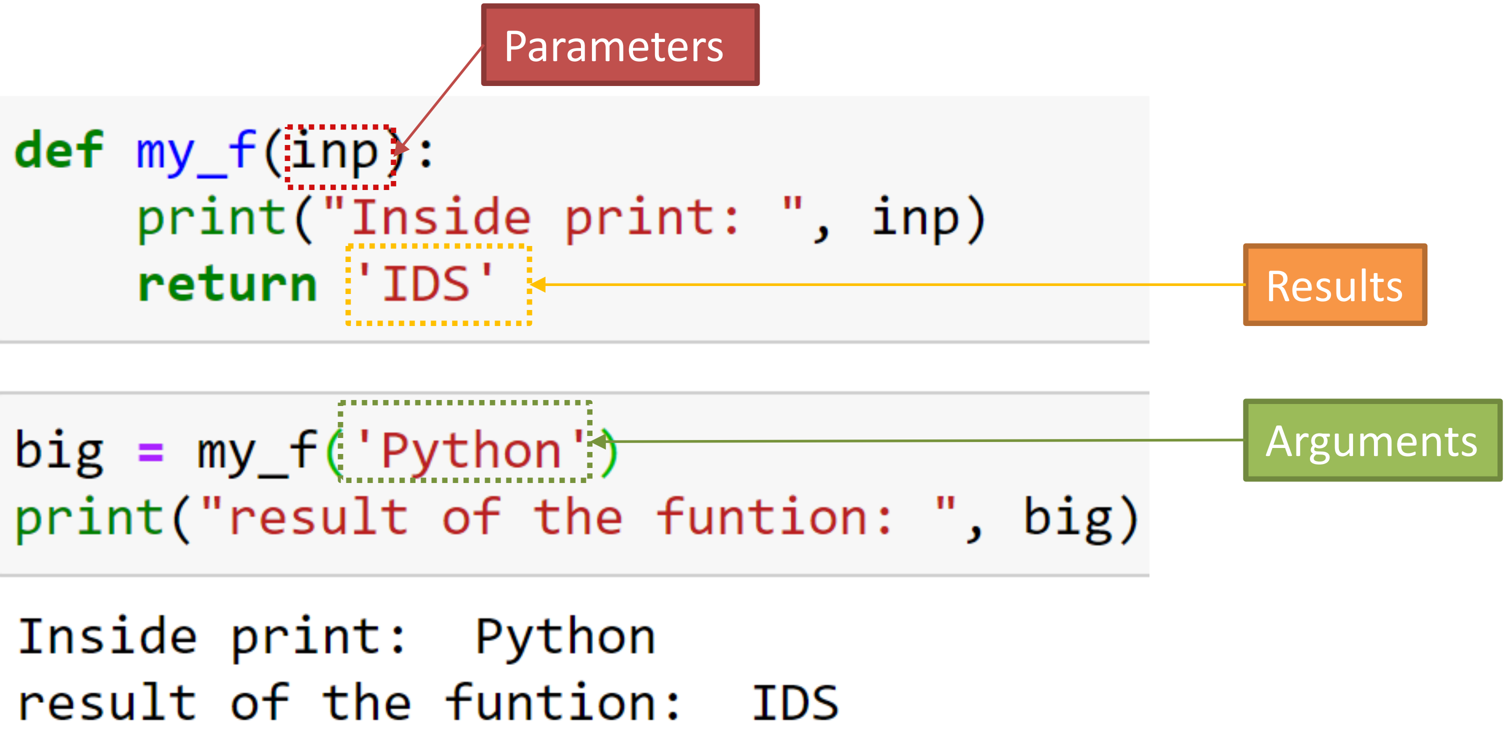


```python
def my_f(inp):
    print("Inside print: ", inp)
    return 'IDS'
```

```python
big = my_f('Python')
print("result of the funtion: ", big)
```

```
Inside print:  Python
result of the funtion:  IDS
```

Arguments, parameters and results



```
def my_f(inp):
    print("Inside print: ", inp)
    return 'IDS'
```

```
big = my_f('Python')
print("result of the funtion: ", big)
```

```
Inside print:  Python
result of the funtion:  IDS
```

Program development

- To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**.
- It is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

Task ***Calculate the distance between two nodes.***

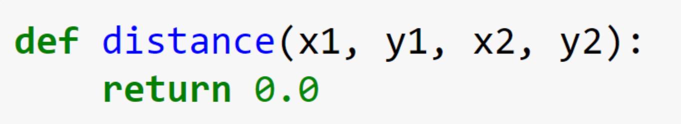
Program development

Task ***calculate the distance between two nodes.***

- **Step 1** Consider what a distance function should look like in Python.

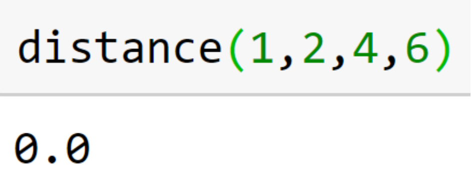
Euclidean distance: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

- **Step 2** What are the inputs (parameters) and what is the output (return value)?



```
def distance(x1, y1, x2, y2):
    return 0.0
```

- **Step 3** To test the function, we call it with sample values.



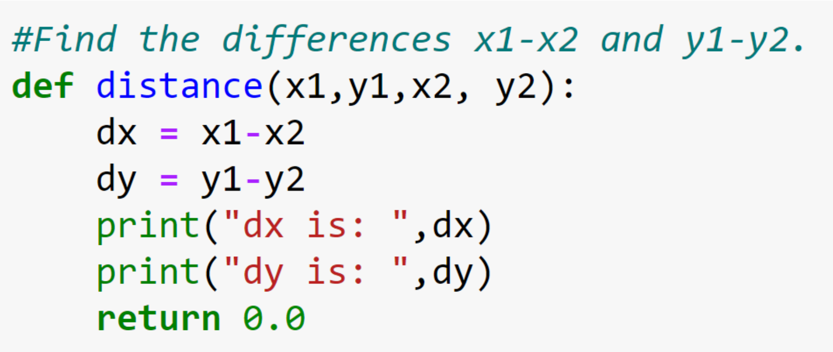
```
distance(1,2,4,6)
```

```
0.0
```

Program development

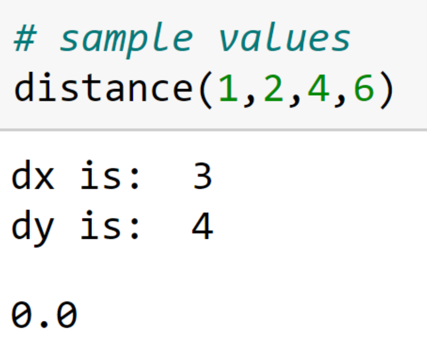
Task *calculate the distance between two nodes.*

- **Step 4** Find the differences x1-x2 and y1-y2.



```
#Find the differences x1-x2 and y1-y2.
def distance(x1,y1,x2, y2):
    dx = x1-x2
    dy = y1-y2
    print("dx is: ",dx)
    print("dy is: ",dy)
    return 0.0
```

- **Step 5** To test the function, we call it with sample values.



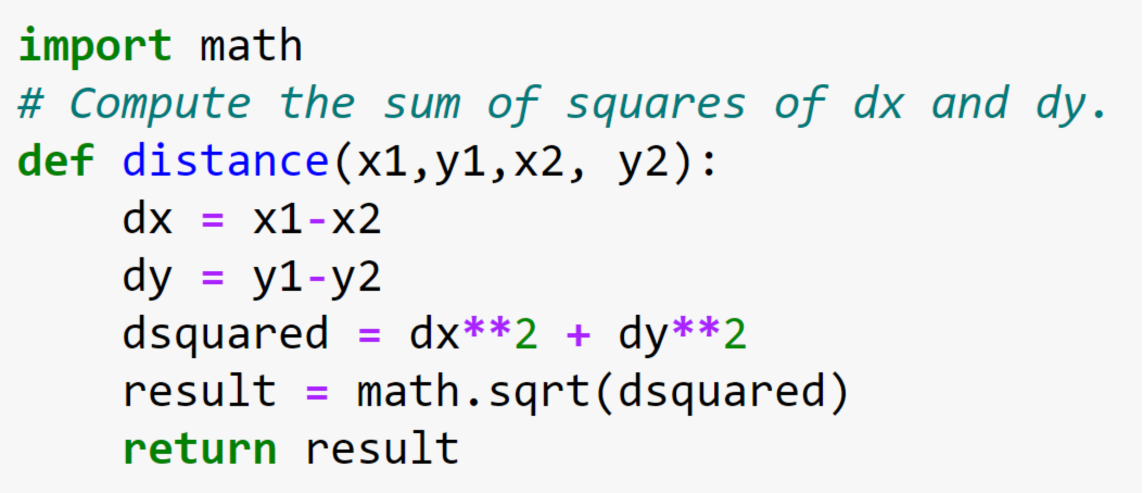
```
# sample values
distance(1,2,4,6)
```

```
dx is:  3
dy is:  4

0.0
```

Program development

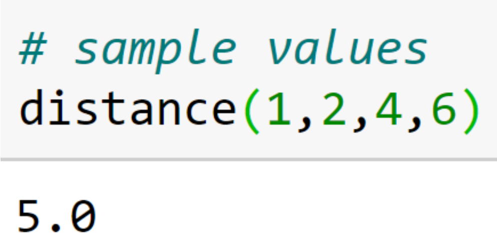
Task *calculate the distance between two nodes.*

- **Step 6** Compute the sum of squares of dx and dy.



```python
import math
# Compute the sum of squares of dx and dy.
def distance(x1,y1,x2, y2):
    dx = x1-x2
    dy = y1-y2
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

- **Step 7** Done!

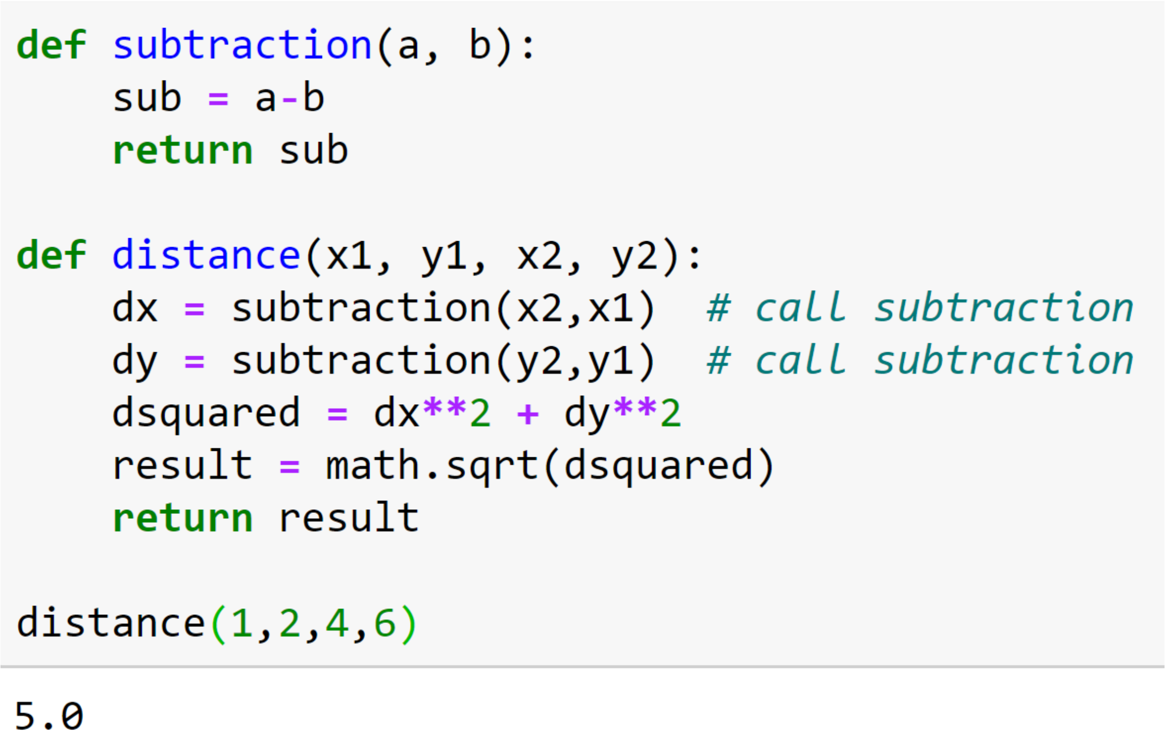


```python
# sample values
distance(1,2,4,6)
```

```
5.0
```

Recursion

- It is legal for one function to call another.

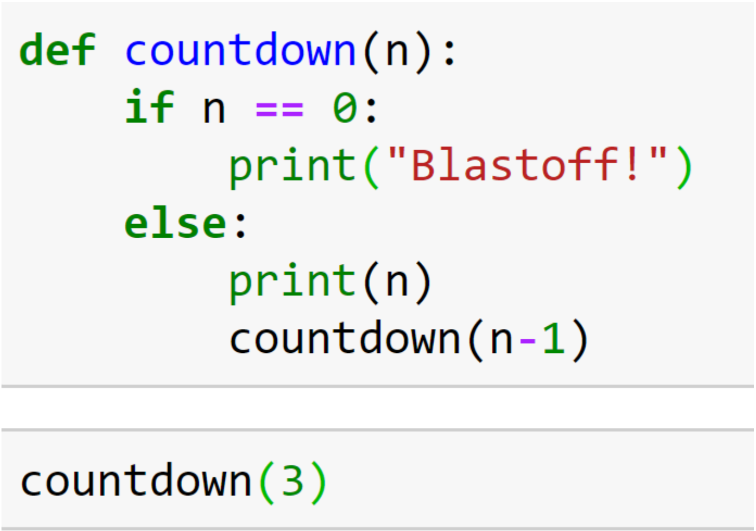


```
def subtraction(a, b):
    sub = a-b
    return sub

def distance(x1, y1, x2, y2):
    dx = subtraction(x2,x1)  # call subtraction
    dy = subtraction(y2,y1)  # call subtraction
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result

distance(1,2,4,6)
```

```
5.0
```

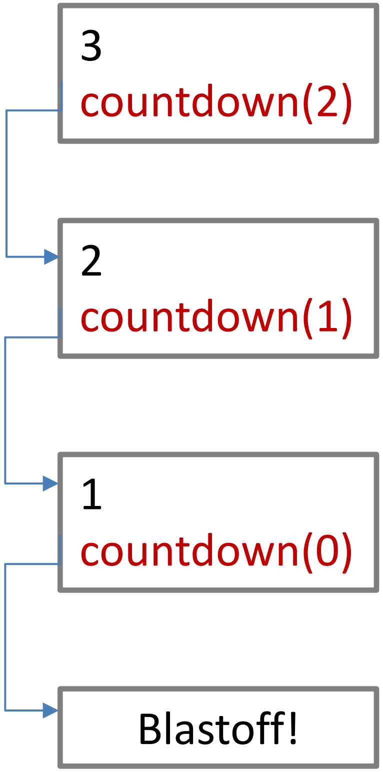
Recursion

Question *What about calling itself?*



```
def countdown(n):
    if n == 0:
        print("Blastoff!")
    else:
        print(n)
        countdown(n-1)
```

```
countdown(3)
```



3
countdown(2)

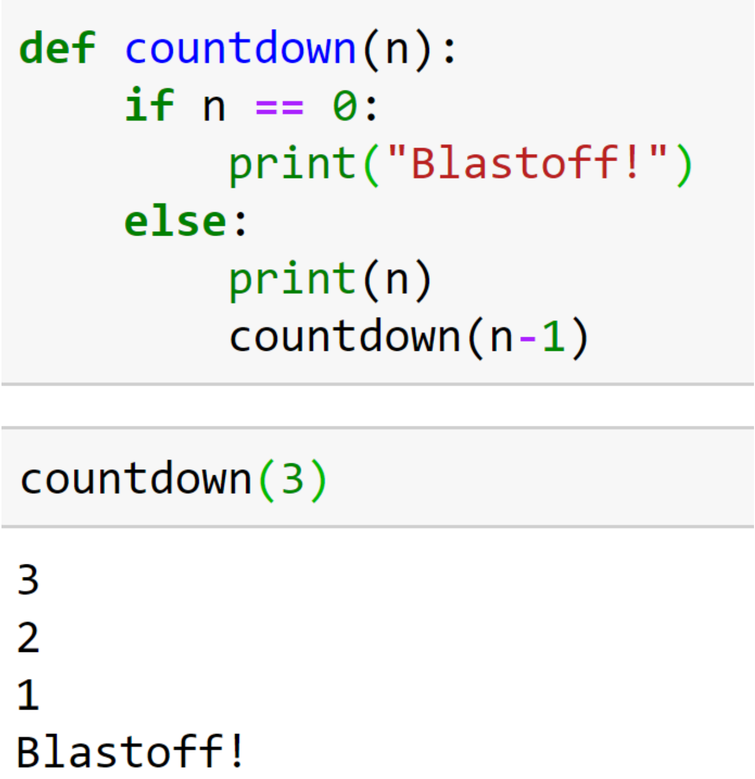
2
countdown(1)

1
countdown(0)

Blastoff!

Recursion

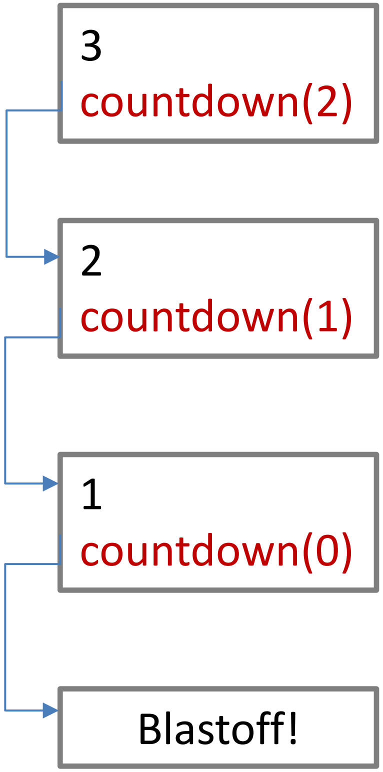
Question *What about calling itself?*



```
def countdown(n):
    if n == 0:
        print("Blastoff!")
    else:
        print(n)
        countdown(n-1)
```

```
countdown(3)
```

```
3
2
1
Blastoff!
```



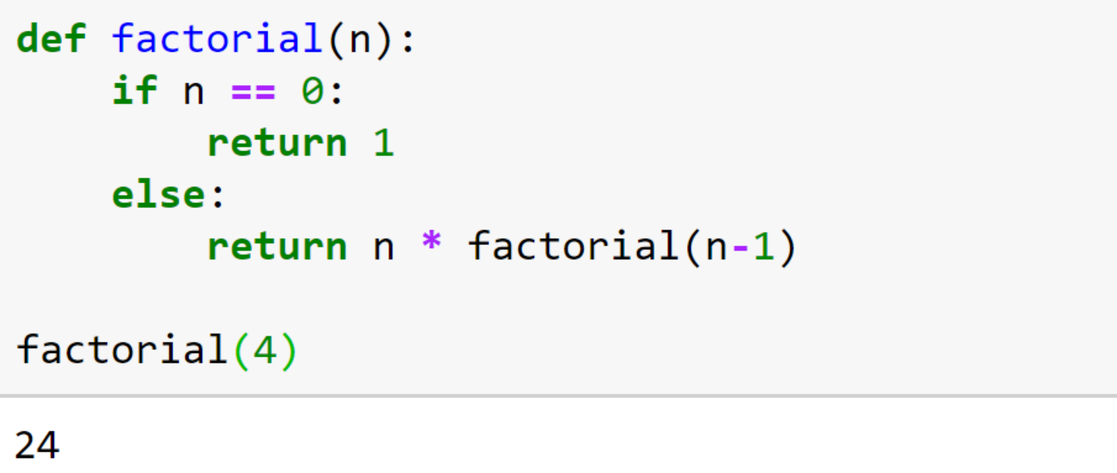
3
countdown(2)

2
countdown(1)

1
countdown(0)

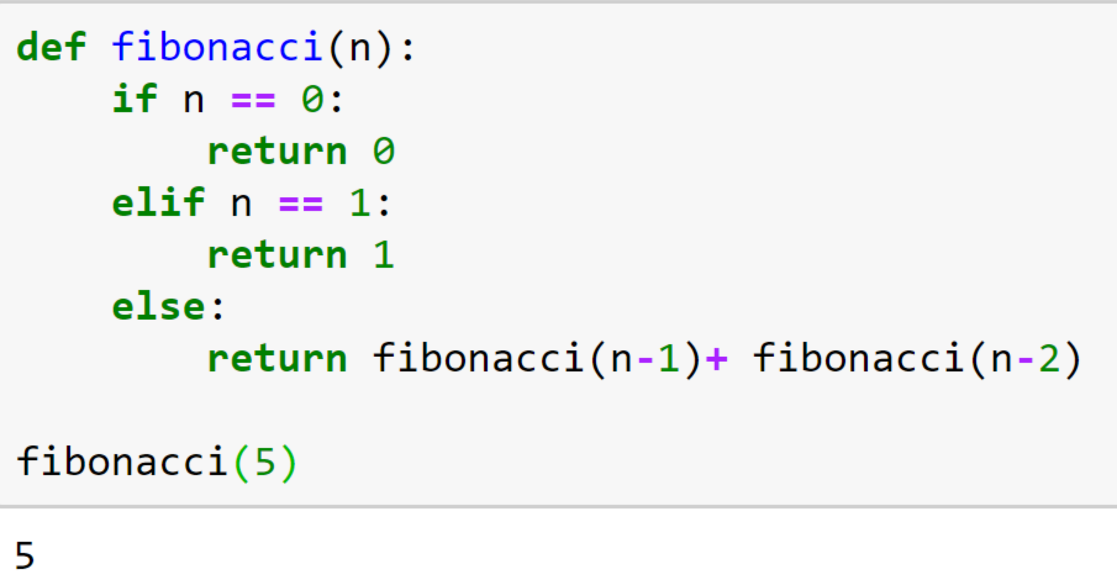
Blastoff!

Examples



```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

factorial(4)
```

```
24
```



```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1)+ fibonacci(n-2)

fibonacci(5)
```

```
5
```

Source: https://realpython.com/fibonacci-sequence-python/

Lab *Functions*