Programming Fundamentals and Python

Data types - Dictionaries

Python Dictionaries

Dictionary in Python is a collection of keys values, used to store data values like a map, which, unlike other data types which hold only a single value as an element. The dictionary is defined into element Keys and values.

- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.

A dictionary is an associative array (also known as hashes). Any key of the dictionary is associated (or mapped) to a value. The values of a dictionary can be any Python data type. So dictionaries are unordered key-value-pairs.

Properties

- **Accessed by key, not offset position**

Dictionaries are sometimes called associative arrays or hashes. They associate a set of values with keys, so an item can be fetched out of a dictionary using the key under which it is originally stored. The same indexing operation can be utilized to get components in a dictionary as in a list, but the index takes the form of a key, not a relative offset.

- **Unordered collections of arbitrary objects**

Unlike in a list, items stored in a dictionary aren‘t kept in any particular order. Keys provide the symbolic (not physical) locations of items in a dictionary.

- **Variable-length, heterogeneous, and arbitrarily nestable**

Like lists, dictionaries can grow and shrink in place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can contain lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a collection of multiple objects if needed, and a given value can be stored under any number of keys.

- **Of the category "mutable mapping"**

Dictionary allows in place changes by assigning to indexes (they are mutable), but they don't support the sequence operations that work on strings and lists. Because dictionaries are unordered collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense.

Instead, dictionaries are the only built-in, core type representatives of the mapping category— objects that map keys to values. Other mappings in Python are created by imported modules.

- **Tables of object references (hash tables)**

If lists are arrays of object references that support access by position, dictionaries are unordered tables of object references that support access by key. Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, dictionaries store object references (not copies, unless explicitly asked).

Create a Dictionary

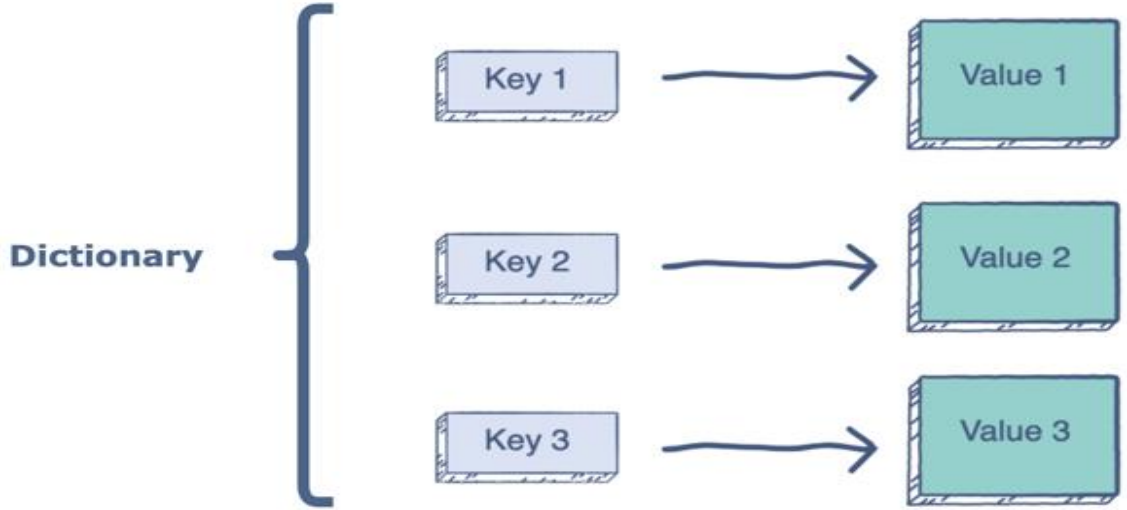
In Python, a dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its Key value. Dictionary can also be created by the built-in function dict( ).

Note – Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.

```
>>> dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
>>> dict
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
>>>
```

How does the dictionary works in Python?

Each key is associated with a single value. The association of a key and a value is called a key-value pair or an item. Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type (i.e., strings, numbers, or tuples).



Dictionary Item

The values in dictionary items can be of any data type. Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

```
>>> dict = {
  "1": "apple",
  "2": False,
  "3": 1964
}
>>> dict
{'1': 'apple', '2': False, '3': 1964}
>>> print(dict['1'])
apple
>>>
```

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change. Unordered means that the items does not have a defined order, you cannot refer to an item by using an index. **Dictionaries cannot have two items with the same key.**

Dictionary Length

To determine how many items a dictionary has, use the **len()** function:

```
>>> dict
{'1': 'apple', '2': 'mani', '3': 1964}
>>> print(dict['1'])
apple
>>> print(len(dict))
3
>>>
```

type()

From Python's perspective, dictionaries are defined as objects with the data type 'dict‘.

```
>>> print(type(dict))
<class 'dict'>
>>>
```

Access

1. The items of a dictionary can be accessed by referring to its key name, inside square brackets.
2. Another method called to access is **get()** that will give you the same result.

```
>>> dict
{'1': 'corolla', '2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus', '6':
'city', '7': '1993'}
>>> x=dict['5']
>>> x
'cultus'
>>> y=dict.get('4')
>>> y
'altis'
>>>
```

Get Keys

The **keys()** method will return a list of all the keys in the dictionary. The list of the keys is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Add a new item to the original dictionary, and see that the keys list gets updated as well.

```
>>> x=dict.keys()
>>> x
dict_keys(['1', '2', '3', '4', '5', '6', '7'])
>>> dict['8']='Alto VXL'
>>> dict
{'1': 'corolla', '2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus', '6':
'city', '7': '1993', '8': 'Alto VXL'}
>>> x
dict_keys(['1', '2', '3', '4', '5', '6', '7', '8'])
>>>
```

Get Values

The **values()** method will return a list of all the values in the dictionary. The list of the values is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list. Make a change in the original dictionary, and see that the values list gets updated, or add a new item to the original dictionary, and see that the values list gets updated as well.

```
>>> a=dict.values()
>>> a
dict_values(['corolla', 'alto', 'farrari', 'altis', 'cultus', 'city', '1993', 'A
lto VXL'])
>>> dict['year']=2022
>>> a
dict_values(['corolla', 'alto', 'farrari', 'altis', 'cultus', 'city', '1993', 'A
lto VXL', 2022])
>>>
```

Get Items

The **items()** method will return each item in a dictionary, as tuples in a list.
The returned list is a view of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list. Make a change or add a new item in the original dictionary, and see that the items list gets updated as well.

```
>>> b=dict.items()
>>> b
dict_items([('1', 'corolla'), ('2', 'alto'), ('3', 'farrari'), ('4', 'altis'), (
'5', 'cultus'), ('6', 'city'), ('7', '1993'), ('8', 'Alto VXL'), ('year', 2022)]
)
>>> dict['year']=1947
>>> b
dict_items([('1', 'corolla'), ('2', 'alto'), ('3', 'farrari'), ('4', 'altis'), (
'5', 'cultus'), ('6', 'city'), ('7', '1993'), ('8', 'Alto VXL'), ('year', 1947)]
)
>>>
```

Check existence of Key

To determine if a specified key is present in a dictionary use the **in** keyword:

```
if "8" in dict:
  print("Yes, it is one of the keys in the dictionary")
else:
    print ('not in the list')
```

```
>>>
==== RESTART: C:\Users\Hera Noor\AppData\Local
Yes, it is one of the keys in the dictionary
>>>
```

Change and Update

You can change the value of a specific item by referring to its key name.

```
>>> dict
{'1': 'corolla', '2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus', '6':
'city', '7': '1993', '8': 'Alto VXL', 'year': 1947}
>>> dict['4']='apple'
>>> dict
{'1': 'corolla', '2': 'alto', '3': 'farrari', '4': 'apple', '5': 'cultus', '6':
'city', '7': '1993', '8': 'Alto VXL', 'year': 1947}
```

The **update()** method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an inerrable object with key:value pairs.

```
>>> dict.update({'4':'altis'})
>>> dict
{'1': 'corolla', '2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus', '6':
'city', '7': '1993', '8': 'Alto VXL', 'year': 1947}
>>>
```

Remove Items

There are several methods to remove items from a dictionary.

- The **pop()** method removes the item with the specified key name
- The **popitem()** method removes the last inserted item (in versions before 3.7, a random item is removed instead).
- The **del** keyword removes the item with the specified key name. The del keyword can also delete the dictionary completely.
- The **clear()** method empties the dictionary.

```
>>> dict
{'1': 'corolla', '2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus', '6':
'city', 'color': 'green'}
>>> dict.pop('color')
'green'
>>> dict.popitem()
('6', 'city')
>>> del dict['1']
>>> dict
{'2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus'}
>>> dict.clear()
>>> dict
{}
```

Copy Dictionary

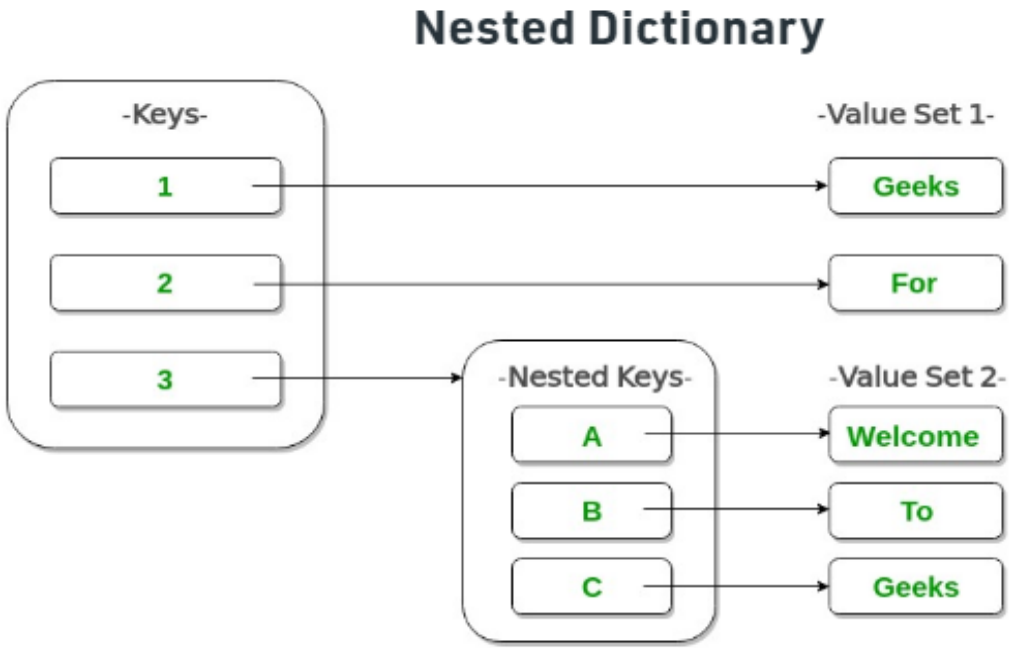
You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

The way to make a copy is to use the built-in Dictionary method **copy().**

```
>>> dict
{'2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus'}
>>> mydict=dict.copy()
>>> mydict
{'2': 'alto', '3': 'farrari', '4': 'altis', '5': 'cultus'}
>>>
```

Nested Dictionary

The dictionaries can be nested, like dictionary within dictionary. For this nested keys are defined.



Nested Dictionary

A dictionary can contain dictionaries, this is called nested dictionaries.

```
>>> myfamily = {
  "child1" : {
    "name" : "Zuha",
    "year" : 2009
  },
  "child2" : {
    "name" : "Mohammad Ahmed",
    "year" : 2014
  },
  "child3" : {
    "name" : "Hasan",
    "year" : 2017
  }
}
```

Summary

- **clear()** – Remove all the elements from the dictionary
- **copy()** – Returns a copy of the dictionary
- **get()** – Returns the value of specified key
- **items()** – Returns a list containing a tuple for each key value pair
- **keys()** – Returns a list containing dictionary's keys
- **fromkeys()** – Returns a dictionary with the specified keys and value
- **pop()** – Remove the element with specified key
- **popitem()** – Removes the last inserted key-value pair
- **update()** – Updates dictionary with specified key-value pairs
- **values()** – Returns a list of all the values of dictionary
- **setdefault()** – Returns the value of the specified key. If the key does not exist: insert the key, with the specified value.
- The **del** keyword removes the item with the specified key name.

Functions

Functions

A function is a groups of statements made to execute them more than once in a program. A function has a name.

Functions can compute a result value and can have parameters that serve as function inputs which may differ each time when function is executed.

Functions are used to:

- Reduce the size of code as it increases the code reusability
- Split a complex problem in to multiple modules (functions) to improve manageability

The Functions

Sequential codes are easy for small scale programs. It becomes harder to keep track of details when code size exceeds.

Advantages:

- Modularity
- Abstraction
- Code reusability

Disadvantages:

- Switching to a function takes time and memory

Module: independent functionalities in a project that can be implemented and tested independently and that these functionalities can be implicated in single project to perform its task.

A function is similar to sequential code, but in functions we assign name to sequential to call for multiple time.

Abstraction means that implementation level details will be hidden from and only superficial level is know, e.g. print(len). But the code that is written inside the function is hidden.

Functions

A function is a block of code which only runs when it is called.

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

A function can return data as a result.

Components of a function:

- Function code - is something that is written inside function or function definition.
- Function call is mandatory for the execution of function.

Functions

Function Components:

- Function Signature
  - * Function Name
  - *Function Arguments (optional)
- Doc string
- Function Body
- Function Return Statement

Syntax:

```
def function_name (arguments):
        doc string
        body
        return statement
```

Function can be used without argument. E.g. Function of length, it accept argument and then return length of the argument.
Function body is the code.
Return statement if needed.

Arguments

Information can be passed into functions as arguments.
Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. The terms parameter and argument can be used for the same thing: i.e. information that are passed into a function.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Apple")

my_function("Red")
my_function("Yellow")
my_function("Green")
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + '', lname)

my_function("Red", 'apple')
my_function("Yellow", 'banana')
my_function("Green", 'chilli')
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
def my_function(country = "Pakistan"):
  print("I am from " + country)

my_function("Karachi")
my_function("Lahore")
my_function()
my_function("DI khan")
```

```
= RESTART: C:\Users\He
8\def.py
I am from Karachi
I am from Lahore
I am from Pakistan
I am from DI khan
```

Example – 1:

Write a function that takes radius of a circle from user and returns its circumference.

```
def circle (r):
    return 2*3.14*r
```

```
=== RESTART: C:/Users/H
>>> print(circle(5))
15.700000000000001
>>> print(circle(3))
9.42
>>> x=7
>>> print(circle(x))
21.98
>>> r=3
>>> print(circle(r))
9.42
```

Example – 1: Understanding doc function.

```
def circle (r):
    '''returns the circumference of circle'''
    return 2*3.14*r
print(circle.__doc__)
```

Docstrings:

- Python docstrings are the string literals that appear right after the definition of a function, method, class, or module.

Docstring

- Docstring is the documentation for a function.
- Docstrings are associated with the object as their __doc__ attribute. We can later use this attribute to retrieve this docstring
- Doc Strings – enclosed in triple single or double quotes. Defines something that the function is doing.
- Built-in variable __doc__ we can print the value of docstring.
- Diff between doc string and hash sign? Comments will not be executed or accessed by any variable, while doc string is accessible.

Docstring

- Functions should be documented for the function users.
- Docstrings are the documentation for a function.
- Example: Implement a function called circle, which is passed a value for radius and it returns the perimeter of the circle.

```
def circle(r):
""" Arguments: radius of a circle (int)
Returns: perimeter of the circle (float)"""
return 2*3.14*r
```

Example – 2: Modifying example-1 to return area of circle too.

```
File  Edit  Format  Run  Options  Window  Help
def circle (r):
    '''returns the circumference of circle'''
    return r*3.14, 3.14*r**2
print(circle.__doc__)
```

```
=== RESTART: C:/Users/Hera Noor/AppDa
returns the circumference of circle
>>> print(circle(5))
(15.700000000000001, 78.5)
>>> x=7
>>> print(circle(x))
(21.98, 153.86)
>>> r=3
>>> print(circle(r))
(9.42, 28.26)
>>>
```

Note: The return statement is tuple as it is separated by comma.

Functions

Example – 3: Modifying example-2 to return area of circle via variables.

```
def circle (r):
    area=3.14*r**2
    circumference=r*3.14
    return area, circumference
```

```
=== RESTART: C:/Users/Hera Noor/AppData
>>> print(circle(5))
(78.5, 15.700000000000001)
>>> x=7
>>> print(circle(x))
(153.86, 21.98)
>>> r=3
>>> print(circle(r))
(28.26, 9.42)
>>>
```

We are utilizing variables and variables
have the values which are calculated against the formula provided by the programmer.

Example – 4: Accessing values of tuples.

```
>>> result=circle(5)
>>> print('area=', result[0],'and',
'circumference=',result[1])
area= 78.5 and circumference= 15.700000000000001
>>>
```

Return function is returning two variables stored in result.

Result is returning two values [0] value which is at zeroth index of return tuple as same for [1].

Here, result is in tuple format, so we are mapping the result value by [0] and [1].

Example – 5: Modifying example-4 to print area and circumference of circle too.

```
def circle (r):
    print('circumference=',r*3.14)
    print('area=',3.14*r**2)
```

```
>>> r=5
>>> circle(r)
circumference= 15.700000000000001
area= 78.5
>>> circle(7)
circumference= 21.98
area= 153.86
>>>
```

Help function

The help function is used to view documentation

```
>>> help(circle)
Help on function circle in module __main__:

circle(r)

>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

>>> help(int)
```

Squeezed text (266 lines).

Argument Types

- Arguments or parameters are the information passed to a function.
- Arguments are specified after the function name, inside the parentheses.
- A function can have as more than one arguments, separated with commas.
- There are two type of arguments in Python:
  - Positional Arguments
  - Keyword Arguments

Argument Types

Positional Arguments

- Must be put in correct order when calling the function.
- Do not have any default value called args.

Keyword Arguments

- Include a keyword and equal sign.
- Used to set defaults; have some default value.
- Placed after the positional arguments if any.
- Can be placed in any order among other keyword arguments when calling the function. called kwargs.

Arbitrary Arguments, *args

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

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

```
def my_function(*kids):
  print("The youngest child is " + kids[0])

my_function("Hadi", "Rahim", "Tina")
```

Keyword Arguments

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

This way the order of the arguments does not matter.

The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child1)

my_function(child1 = "Hadi", child2 = "Wara", child3 = "Rahim")
```

Arbitrary Keyword Arguments, **kwargs

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

This way the function will receive a dictionary of arguments, and can access the items accordingly. If the number of keyword arguments is unknown, add a double ** before the parameter name.

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Hadi", lname = "Rehman")
```

Variable Length Argument - **kwargs

Example:

```
def my_func(**kwargs):
    for item in kwargs.items():
        print (item)
my_func(first ='Computer', last='Programming')
```

```
= RESTART: C:\Users\Hera No
\define func namespace.py
('first', 'Computer')
('last', 'Programming')
>>>
```

VLA – Variable Length Argument

- In Python, variable number of arguments can be passed to a function.

Example: Consider the min() function.

min(3,5,66)
min(3,4,2,6,44,1,0,88,6)

} Both are correct.
- min() is a VLA function

- Special symbols are used for this purpose.
  - *<args_name> or *args: for positional / non-keyword arguments.
  - **<argument_name> or **kwargs: for keyword arguments.

VLA *args

Example: The following code used a function to print all objects passed to it, on separate lines.

CODE

```
def my_print(*args):
    for value in args:
        print(value)
my_print('Learning','Python','3')
print()
my_print('Learning','Python','3','and','Java','too')
```

OUTPUT

```
\define func namespace
Learning
Python
3

Learning
Python
3
and
Java
too
>>>
```

Module, Packages and Libraries

- A Python module is a py file containing functions and classes.
  - useful code that eliminates the need for writing codes from scratch.

- A package or library is a collection of modules.

- Python has a rich set of libraries.
  - There are over 137,000 python libraries present today.

- Python libraries play a vital role in developing machine learning, data science, data visualization, image and data manipulation applications and more.

Builtin Modules

- It contains all the core built-in types/classes and functions.
- Loaded automatically when the interpreter starts.
- It does not contain all the functions and classes supported by Python.
  - size is kept small for efficiency.
- Some functions of builtins module are:
  - min()
  - max()
  - len()
  - print()
- Other modules are kept in the library called The Standard Library.

The Standard Library

- It is the native library of Python that comes with Python installation.
- The modules of this library need to be imported if used.
- Some modules in The Standard Library include:
    - mathematical functions
    - pseudorandom generators
    - fraction handling
    - date-time functions
    - graphical user interface (GUI) development

See https://docs.python.org/ for details on builtins module and the Standard Library.

Third Party Library

- Python has a huge collection of third-party modules/packages available for usage.
- These modules/packages must be installed and then imported if used.
- Some commonly used Python third party packages:
    - Pillow: for handling different formats of images
    - Matplotlib: for two dimensional graphs and plots
    - Numpy: for array processing
    - Scipy: for scientific and technical computation
    - Pandas: to organize, explore, represent, and manipulate data
- Package installers:
    - pip installs Python packages
    - conda installs packages written in any language

The import Statement

- When the Python interpreter executes an import statement:
  - the corresponding file containing the module is searched.
  - the module code is run to create objects defined in that module.
  - a namespace is created where the objects live.

- Hence a separated namespace is created for every module.

- An application may consist of several modules:
  - top level module is the main program from which the execution starts. It is called main.
  - the remaining modules are library modules that are imported by the top level module.

How to import…

1. Using import statement

Creates a separate namespace

Example:

```
>>> import math
>>> math.sqrt(49)
```

Single statement can be used to import more than one module.

```
>>> import math,random
```

How to import…

2. Using from with import

- It copies selected functions into the namespace of the importing module.
- Remaining functions are not instantiated.
- No separate namespace is created.
- Functions can be called without the dot operator.

Example:

```
>>> from math import sqrt
>>> sqrt(49)
```

How to import…

3. Using from to import all functions of a module

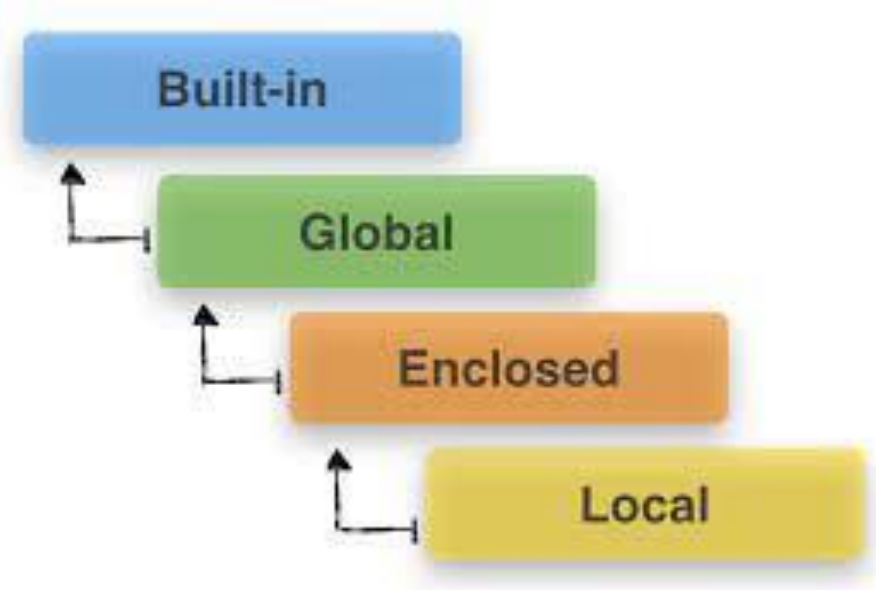
Example:

```
>>> from math import *
>>> sqrt(49)
```

- Now all the functions are copied to the calling namespace and can be used without the dot operator.
- Not a good idea!!
  - Takes too much space
  - Variable names may clash

Python Scope

We have already gone through all the scope level in the previous class, lets have some more examples.



The LEGB Rule

The Python interpreter searches a name (variable of function) in the following order:

1. Search the local namespace – L
2. Search the enclosing function's namespace (for nested functions) – E
3. Search the global namespace – G
4. Search the namespace of module builtins – B

The builtin module is automatically loaded every time Python interpreter starts.

- It is the top level execution environment.
- It provides direct access to all built-in identifiers (types/classes and functions) of Python.

The LEGB Rule

Example:

```
def f(b):
    a=b
    return a*b

x=3
print(f(x))
print(x)
```

- `f` has global namespace as it is defined in the main module.
- `a` and `b`, defined in `f()` have local scopes.
- `x` defines in main module has global namespace.
- `print()` function is defined in builtins modules.

ns_main (global)	ns_f (local)	ns_print (module builtins)
f	a	print
x	b	

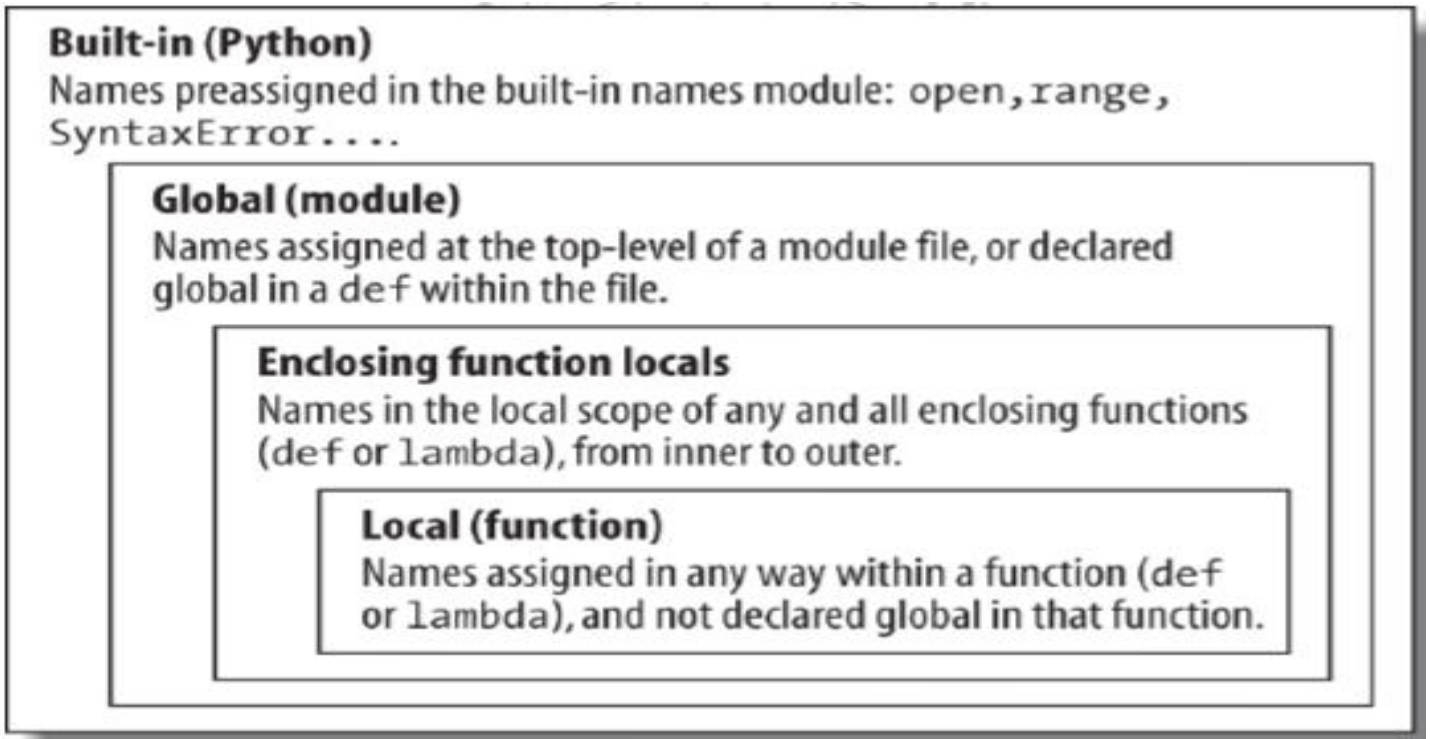
Output

9

3

Scope Example

Let's step through a larger example that demonstrates scope ideas. Suppose we wrote the following code in a module file:



Python Scope Basics

- Names assigned inside a def can only be seen by the code within that def. You cannot even refer to such names from outside the function.
- Names assigned inside a def do not clash with variables outside the def, even if the same names are used elsewhere. A name X assigned outside a given def (i.e., in a different def or at the top level of a module file) is a completely different variable from a name X assigned inside that def.
- If a variable is assigned inside a def, it is local to that function.
- If a variable is assigned in an enclosing def, it is nonlocal to nested functions.
- If a variable is assigned outside all defs, it is global to the entire file.

The Global keyword

The keyword called global used inside a function allows the function to update a global variable.

Practice Problem: Modifying the last code on the previous slide.

```
def h():
    global x
    x+=1
    print(x)

x=5
h()
print(x)
```

Now this is the same global x accessed from this local namespace

Output

6

6

Global Variable

In general, functions should rely on arguments and return values instead of globals, Although there are times when globals are useful, variables assigned in a def are local by default because that is normally the best policy.

Changing globals can lead to well-known software engineering problems: because the variables' values are dependent on the order of calls to arbitrarily distant functions, programs can become difficult to debug, or to understand at all.

```
def intersect(seq1, seq2):
    res = [] # Start empty
    for x in seq1: # Scan seq1
        if x in seq2: # Common item?
            res.append(x) # Add to end
    return res
```

```
= RESTART: C:\Users\Hera Noor
on\Python38\def.py
>>> seq1='love'
>>> seq2='mango'
>>> intersect(seq1, seq2)
['o']
>>>
```

Use of Global Variables is not Welcomed!

They make behavior of a function/code unpredictable.

Example:

Consider the following functio

```
def process(n):
        return m+n
def inc_m():
        global m
        m+=1
def assign_m():
        global m
        m=5
```

Find outputs of the following main codes

1.
```
process(12)
```
Output: can't say!!

2.
```
assign_m()
print(process(12))
```
Output: 17

3.
```
m=10
 print(process(12))
```
Output: 22

4.
```
m=0
inc_m()
inc_m()
print(process(12))
```
Output: 14

5.
```
assign(m)
inc_m()
inc_m()
print(process(12))
```
Output: 19

Scope – Local vs Global

- Scope refers to the visibility of a variable defined in one namespace to another namespace.
- It simply refers to the lifetime of a variable in a program.
- Every name (variable name or function name) has a scope according to the namespace where it lives.
- Outside of its scope the variable does not exist.

Scope – Local vs Global

Local Scope: names assigned inside a function.
- such variables are called local variables.

Global Scope: names assigned in the interpreter shell or main module (outside of any function).
- such variables are called global variables.

Advantages of local variables

Functional independence – same variable name can be used in different functions.
Does not occupy memory unnecessarily.

Advantages of global variables

Can be used to define constants in a program.

Scope example

Example-1

```
# Global scope
X = 99 # X and func assigned in module: global
def func(Y): # Y and Z assigned in function: locals
# Local scope
        Z = X + Y # X is a global
return Z
func(1) # func in module: result=100
```

Scope example

Example-2

```
X = 88 # Global X
def func():
        global X
        X = 99 # Global X: outside def
func()
print(X) # Prints 99
```

Namespaces

Namespaces

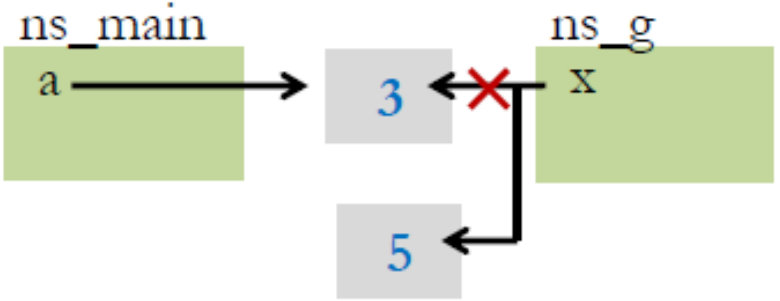
- Namespace is an abstract container or environment created to hold a logical grouping of identifiers / variables.
- An identifier/variable defined in a namespace is usually associated only with that namespace.
- The same identifier/variable can be independently defined in multiple namespaces.
- An identifier/variable defined in one namespace may or may not interfere with the identifier/variable defined in another namespace.
- Languages that support namespaces specify the rules that determine to which namespace an identifier/variable belongs.

Parameter passing in Namespaces

Passing Immutable Objects as Parameters

Example 1: Consider the following code:

```
def g(x):        Step 3
    x=5          Step 4
a=3              Step 1
g(a)             Step 2
print(a)         Step 5
```



Output
3

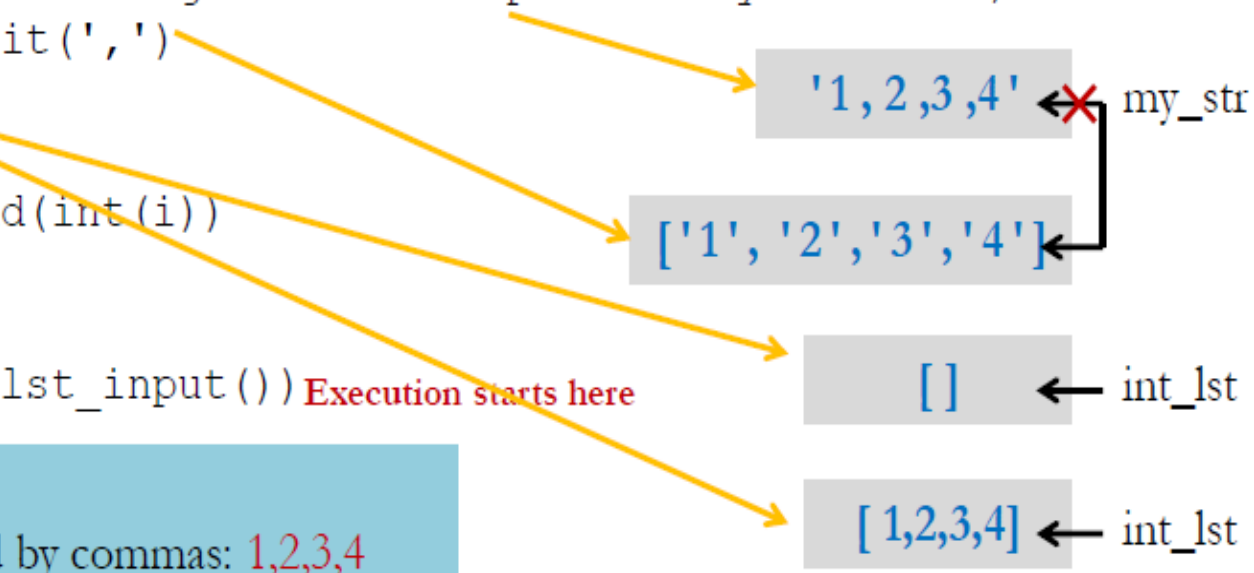
Problem

Write a function that inputs a list of integers from user.

```
def lst_input():
    my_str=input('Enter integer values separated by commas:')
    my_str=my_str.split(',')
    int_lst=[]
    for i in my_str:
        int_lst.append(int(i))
    return int_lst

print('You entered:',lst_input())
```

Execution starts here



Output

Enter integer values separated by commas: 1,2,3,4

You entered: [1,2,3,4]