IT5001 Software Development Fundamentals

2. Functions (Callable Units)

Rajendra Prasad Sirigina

August-2023

Agenda: Functions

- Python Modules and Packages
  - ➢ Namespaces
  - ➢ Scope

- User-defined functions
  - ➢ Arguments
    - o Positional
    - o Keyword
    - o Default
  - ➢ Return Statement
    - o print vs return
  - ➢ Function Tracing
    - o Namespaces
    - o Local namespace

- Pure Functions

Why Functions?

Example: Drawing a square

```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```
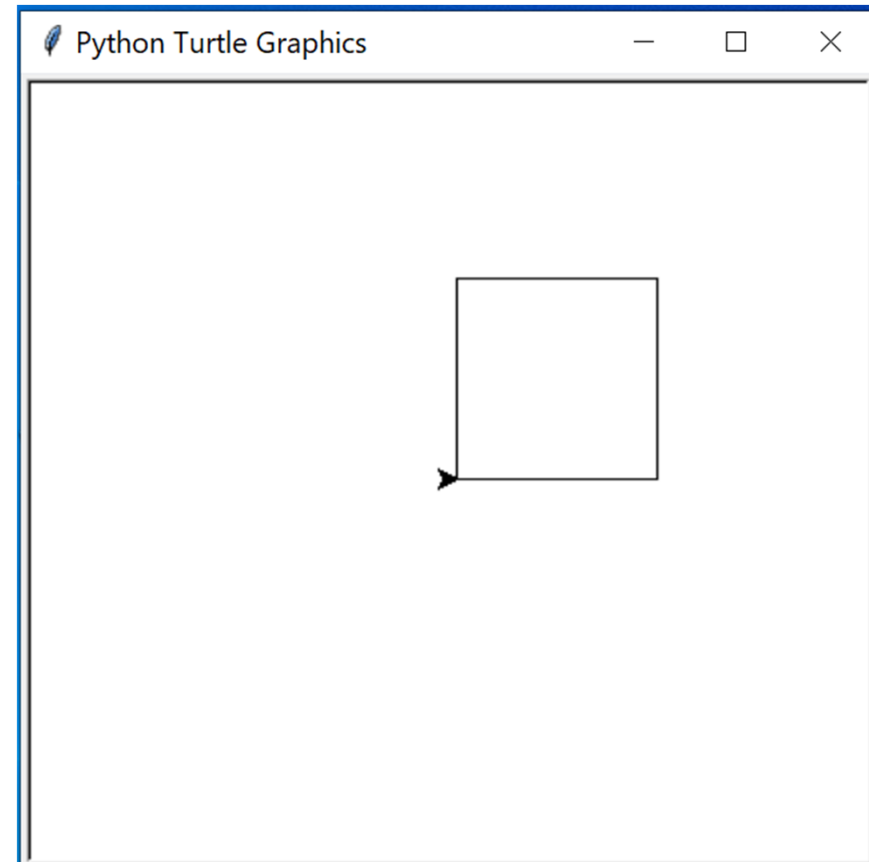

What if we need to draw another square at a different location?

Drawing another square

```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```
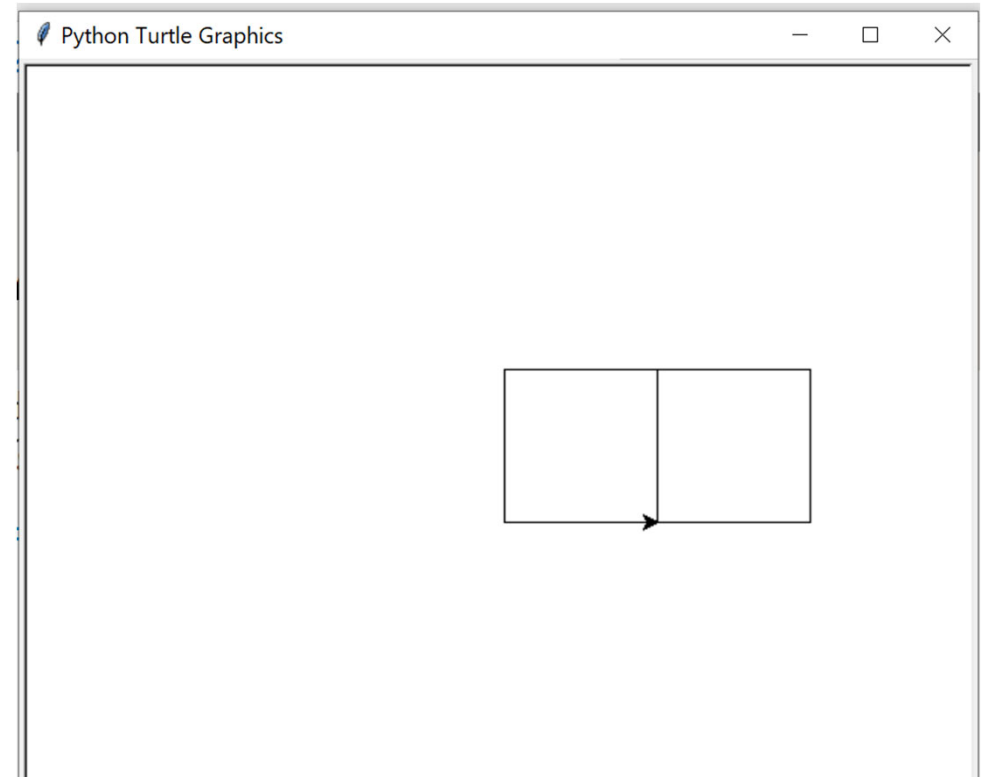
What if you want more squares?

- Involves a lot of repetitions

- Instead write code to draw a square and reuse it

- How to do it?
  - ➢ Use functions

- Functions allows reusability
  - ➢ No need to rewrite the code for same functionality

Abstraction
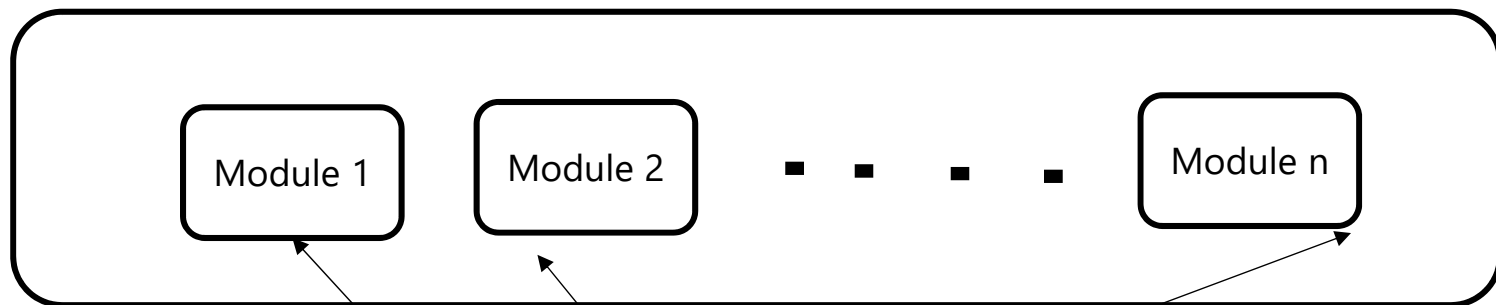
Input → $function$ → Output
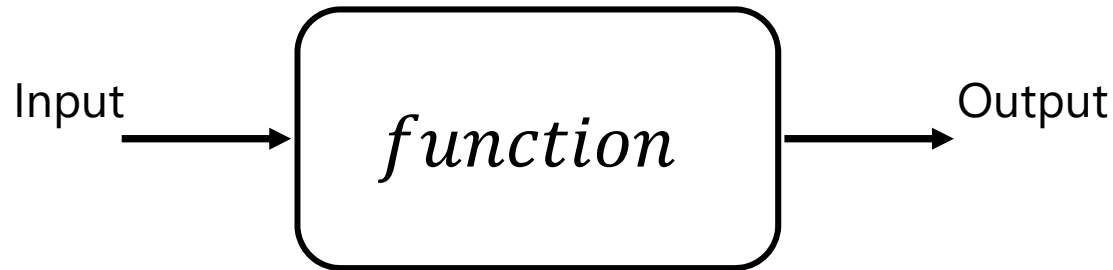
- Usability
  - ➢ User only needs to know input arguments and output data types

- Declarative style of programming
  - ➢ Tell the system what you want

Modularity

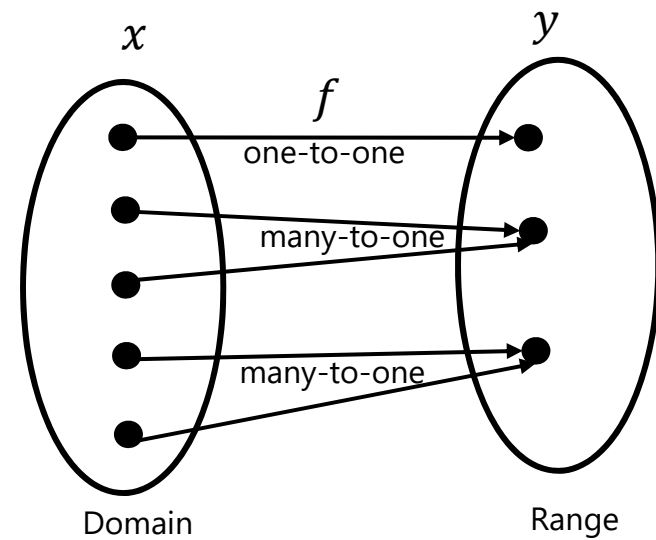A Giant Module
(A program that do many things)

Vs

Modular Architecture

Module 1   Module 2   ▪ ▪ ▪ ▪   Module n

programs that only do one thing

Functions: Mathematics

- A mapping from input to output

name of the function

$$y = f(x)$$

output   argument

x y

f

one-to-one
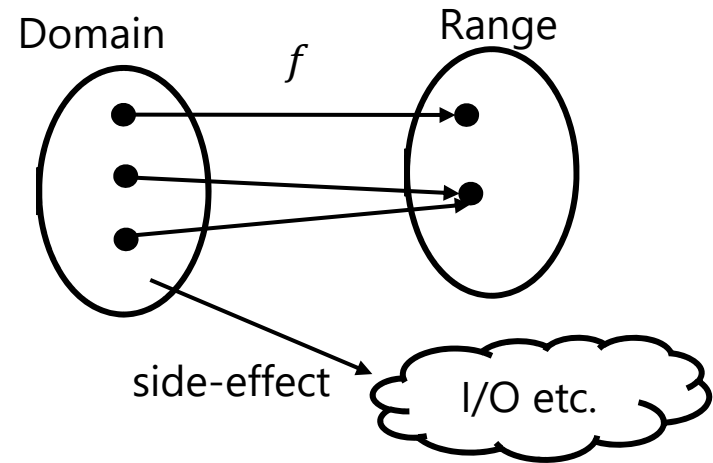
many-to-one

many-to-one

Domain   Range
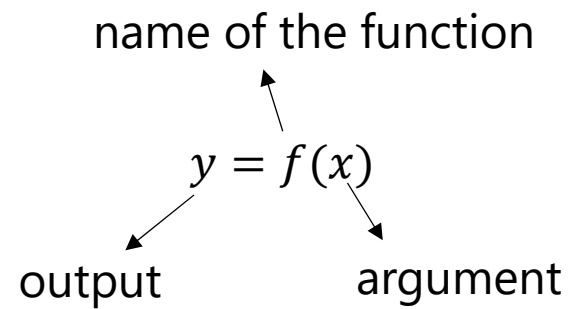
Functions in Programming

- A self-contained block of code

- Known as subroutines, methods, procedures, etc.

Types of Functions

- Pure functions
  - ➤ Functions without side-effects
  - ➤ Only mapping
  - ➤ Outputs depend only on inputs

- Functions with side-effects
  - ➤ Side-effects: I/O tasks
    - ○ Taking inputs from keyboard, reading data from a file, etc.
    - ○ Printing output to screen, writing data to a file, etc.

- Higher-order Functions (in Week 4)
  - ➤ Functions that accept functions as inputs and/or return functions as outputs

$$h(x) = f(g(x))$$

Functions in Python

- Built-in Functions
  - Builtin module is loaded automatically
    - No need to import
  - Examples: print(), input(), id(), etc.

- Functions from Python modules and packages
  - Not loaded automatically
  - Need to import them when needed
  - Example: function in modules like math, statistics.

- User-defined Functions

- Special function
  - main()

Built-In Functions

Built-in Functions

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		**_**
	iter()		__import__()

https://docs.python.org/3/library/functions.html

Functions from Python Modules and Packages

- Import packages
  - ➢ Example:
    - ○ math package

- We can import modules
  - ➢ Syntax
    - ○ import *<module_name>*
    - ○ from *<module_name>* import *<name(s)>*

  - ➢ Examples
    - ○ import math
    - ○ from math import cos, sin
    - ○ from math import *

```
import math
x = math.pi/2
y = math.sin(x)
print(y)
```

```
from math import sin, pi
x = pi/2
y =sin(x)
print(y)
```

Namespaces: Importing Modules
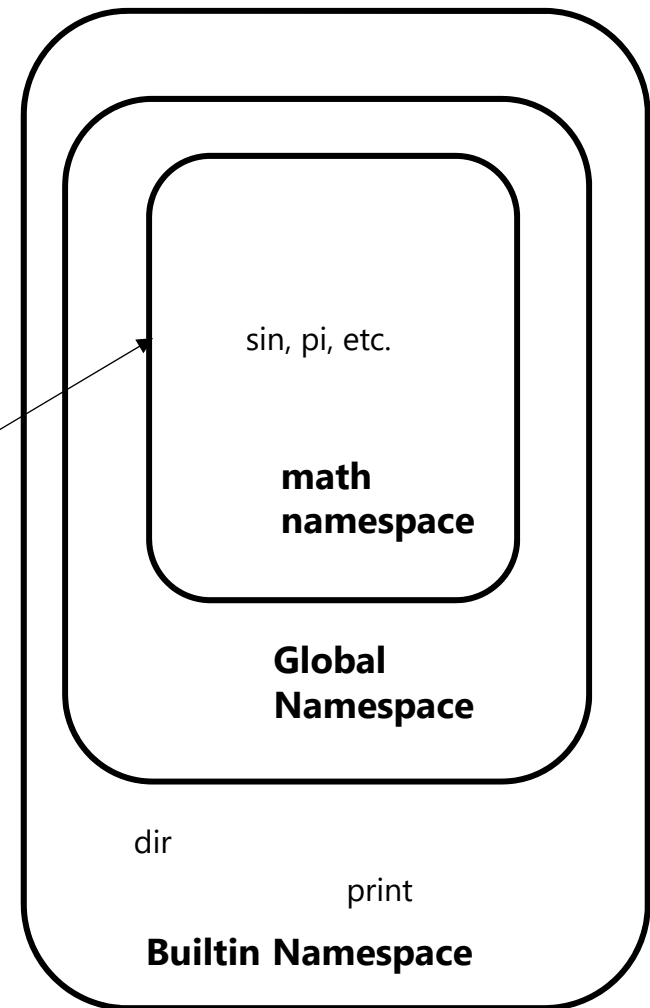
Syntax:

import *<module_name>*

<module_name>.<object_name>

Example:

```
import math
math.pi
math.sin(math.pi/2)
```

```
>>> dir(__builtins__)
>>> print(globals())
>>> dir(math)
```

dedicated
namespace
for each
imported
module within
global
namespace

sin, pi, etc.

math namespace

Global Namespace

dir

print

Builtin Namespace

Importing Modules

```
>>> dir(math)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    dir(math)
NameError: name 'math' is not defined
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <cl
ass '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {
}, '__builtins__': <module 'builtins' (built-in)>}
>>> import math
>>> print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <cl
ass '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {
}, '__builtins__': <module 'builtins' (built-in)>, 'math': <module 'math' (built
-in)>}
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh'
, 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expml', 'fabs', 'factorial', 'floor', 'fm
od', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'is
inf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan'
, 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'ta
n', 'trunc']
```
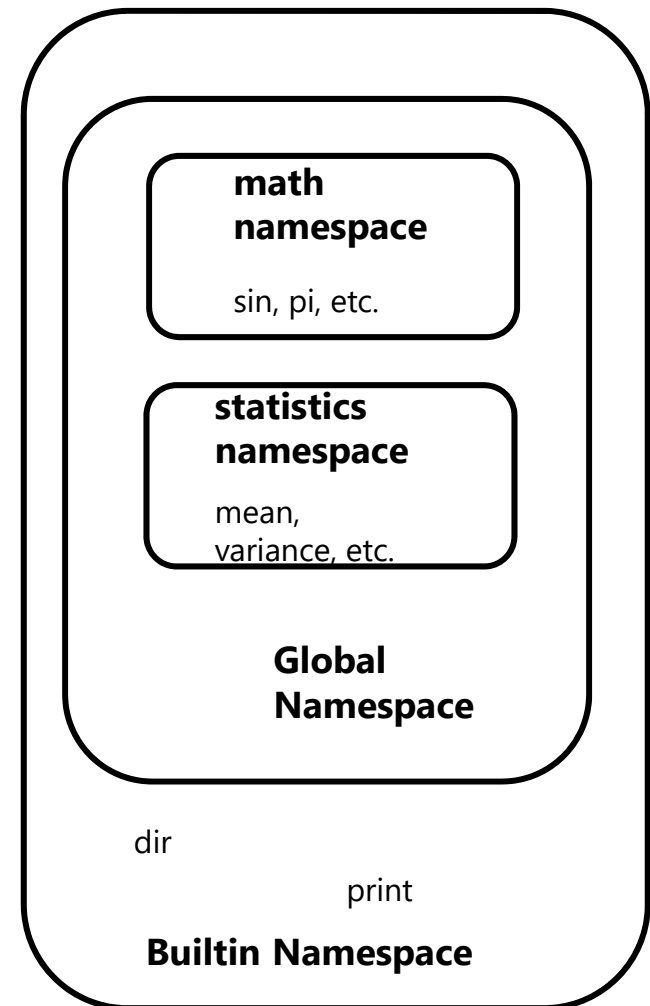
Namespaces: Importing Multiple Modules

Syntax:

import *<module_1>*, *<module_2>*

<module_1>.<object_name>
<module_2>.<object_name>

Example:

```python
import math, statistics
sin_pi_by_2 = math.sin(math.pi/2)
mean_ = statistics.mean([1,2])
print(f'sin_pi_by_2 is {sin_pi_by_2}')
print(f'Mean of 1 and 2 is {mean_}')
```

math namespace

sin, pi, etc.

statistics namespace

mean, variance, etc.

Global Namespace

dir

print

Builtin Namespace

17

Namespaces: Importing Objects from Modules

Syntax:
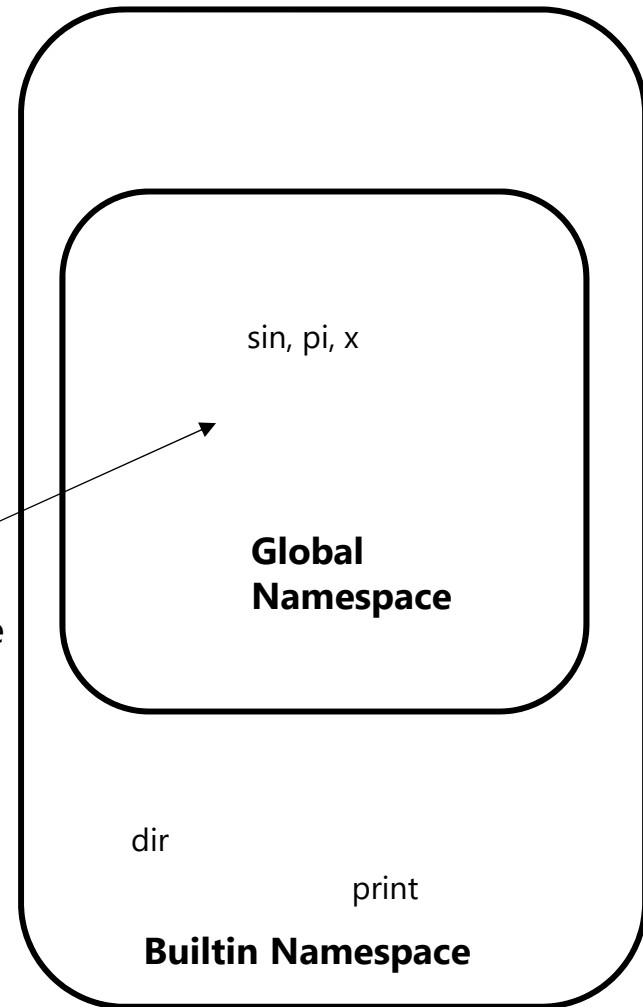
from *<module_name>* import *<object_name>*

<module_name>.<object_name>

Example:

```python
from math import sin, pi
print(globals())
x = sin(pi/2)
print(f'sin_pi_by_2 is {x}')
```

Names from math module are imported into global namespace

sin, pi, x

Global Namespace

dir

print

Builtin Namespace

18

You don't want to be like this in real life

Try NOT to import too many packages

User-defined Functions

- Simple functions
- Doc strings
- Arguments (Parameters)
  - ➢ Positional
  - ➢ Keyword
  - ➢ Default
- Parameter Passing
  - ➢ Pass-by Value
  - ➢ Pass-by Reference
  - ➢ Pass-by Assignment

User-Defined Functions

function name

keyword

<statement>
<statement>
def function_name (*argument(s)*):
    <statement>
    <statement>
    .
    .
    .
    <statement>
<statement>
<statement>

white space
(indicates the scope of the function)
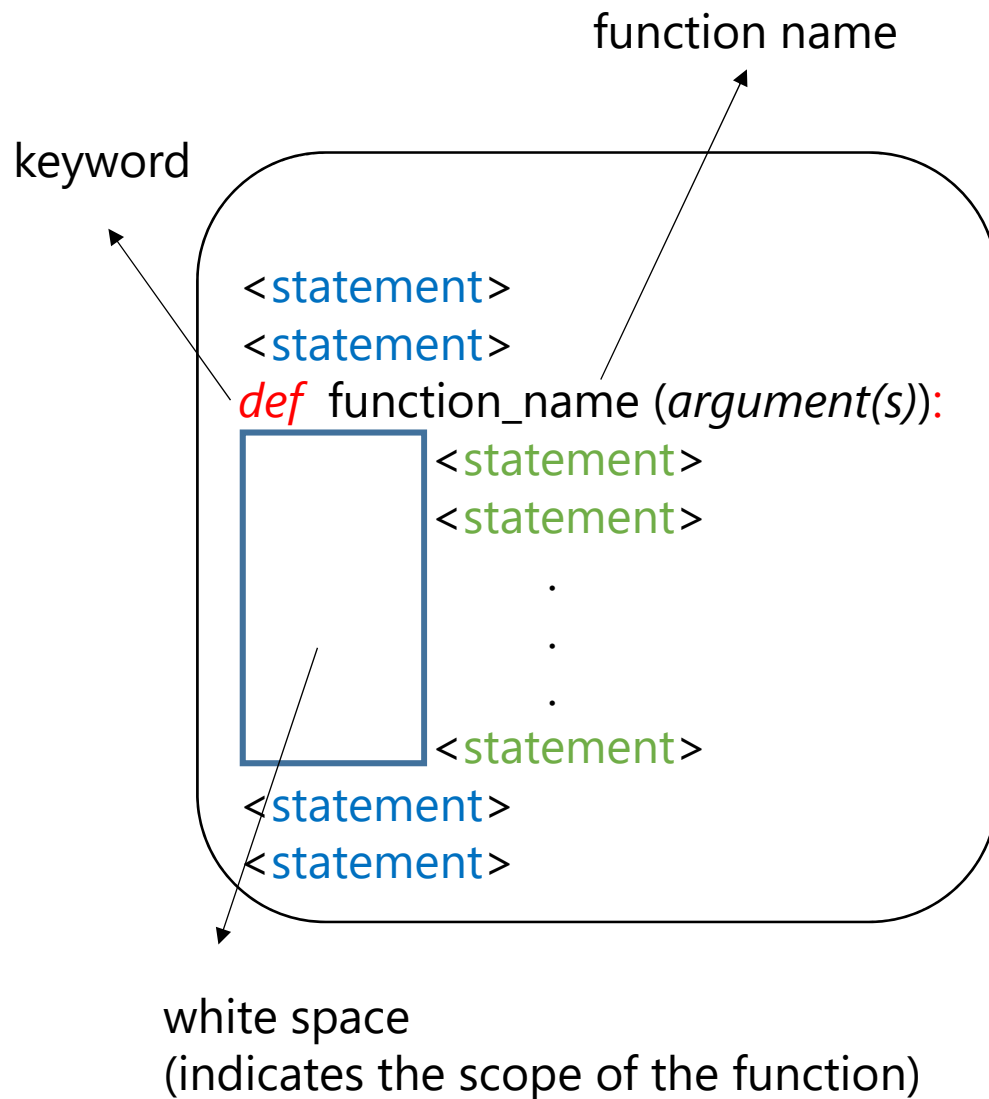
y = function_name(*argument*)

<statement>: Statement that is part of the function scope

<statement>: Statement that is out of function scope

User-Defined Functions

```python
print("Hello!")
def my_function_1():
    print("Hello Functions!")

    print("I am also a part of the function")
print("I am not a part of the function")

my_function_1()
```

Return Statement

- It does two things
  - ➢ Terminates the function
  - ➢ Return statement pass the output of function to the calling function

def function_name (*arg_1, arg_2,..., arg_n*):
        <statement>
        <statement>
          .
          .
          .
        return <statement>

Example

$$y = f(x_1, x_2) = \frac{x_1 + x_2}{2}$$

name of function   arguments

Block of Code

```python
def average(number_1, number_2):
    sum_of_numbers = number_1+number_2
    average_of_numbers = sum_of_numbers/2
    return average_of_numbers
```

keywords   white space   Value returned by the function
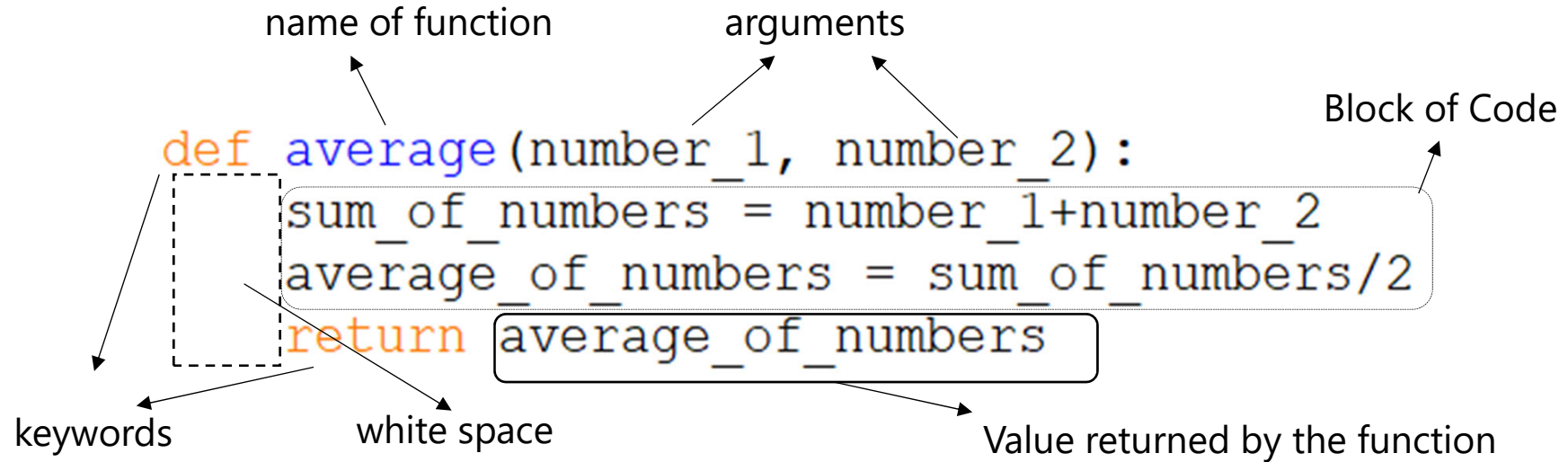
Arguments

- Provide input to functions

def function_name (*arg_1, arg_2,...,arg_n*):
    \<statement\>
    \<statement\>
     .
     .
     .
    \<statement\>

name of the function

$$y = f(x_1, x_2, .., x_n)$$

output

arguments

```python
def my_module(arg_1):
    print(f'Module: {arg_1}')
my_module('IT5001')
```

Doc String

- Contains information about function
  - ➢ Describes how to use the function
  - ➢ Can access it using help/doc methods

```python
def my_module(arg_1):
    '''
    Parameters:
        arg_1 (str): ID of the module
                     Must be a string
    Returns:
        This function returns nothing

    Example:
        module_ID = 'IT5001'
        my_module(module_ID)
    Output:
        Module: IT5001
    '''
    print(f'Module: {arg_1}')
```

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

my_module(arg_1)
    Parameters:
        arg_1 (str): ID of the module
                     Must be a string
    Returns:
        This function returns nothing

    Example:
        module_ID = 'IT5001'
        function_2(module_ID)
    Output:
        Module: IT5001
```

Types of Arguments

- Positional arguments

- Keyword arguments

- Default/optional arguments

Positional Arguments

```python
def module_info(module_code, module_name, module_type):
    print(f"{module_code}: {module_name} is an {module_type} Module")


>>> module_info("IT5001", "Software Development Fundamentals", 'Essential')
IT5001: Software Development Fundamentals is an Essential Module
```

Positional Arguments

- Number of arguments are important

```python
def module_info(module_code, module_name, module_type):
    print(f"{module_code}: {module_name} is an {module_type} Module")
```

```
>>> module_info("IT5001", "Software Development Fundamentals")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    module_info("IT5001", "Software Development Fundamentals")
TypeError: module_info() missing 1 required positional argument: 'module_type'
```

- Should take care of order of arguments

```
>>> module_info("IT5001", 'Essential', "Software Development Fundamentals")
IT5001: Essential is an Software Development Fundamentals Module
```

Keyword Arguments

- Order is not important

```python
def module_info(module_code, module_name, module_type):
    print(f"{module_code}: {module_name} is an {module_type} Module")

>>> module_info(module_code = "IT5001", module_type =  'Essential', module_name =
"Software Development Fundamentals")
IT5001: Software Development Fundamentals is an Essential Module
```

- Names of the arguments should be the same as in function definition

```python
>>> module_info(module_code = "IT5001", module_type =  'Essential', module__name =
"Software Development Fundamentals")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    module_info(module_code = "IT5001", module_type =  'Essential', module__name =
"Software Development Fundamentals")
TypeError: module_info() got an unexpected keyword argument 'module__name'
```

Default/optional arguments

- Assign default values in function definition

default argument should always be at the end

```python
def module_info(module_code, module_name, module_type = 'Essential'):
    print(f"{module_code}: {module_name} is an {module_type} Module")
```

- Default arguments can be omitted while calling a function

```python
>>> module_info("IT5001", "Software Development Fundamentals")
IT5001: Software Development Fundamentals is an Essential Module
```

- Default argument can be assigned a different value

```python
>>> module_info("CS522", " Advanced Computer Architecture", module_type  = 'Elective')
CS522:  Advanced Computer Architecture is an Elective Module
```

Return vs Print

```
def sum_two_numbers(arg_1, arg_2):
    return arg_1 + arg_2
```

Vs

Returns None

```
def sum_two_numbers(arg_1, arg_2):
    print(arg_1 + arg_2)
```

```
x = sum_two_numbers(2,3)
```

Revisit Drawing Squares

- How to generalize the code?

```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```

- Retain similarities
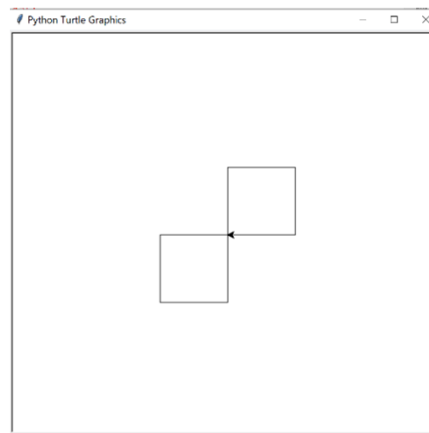- Parametrize differences

Drawing Squares using Functions

```python
from turtle import *
def draw_square(side_length):
    forward(side_length)
    left(90)
    forward(side_length)
    left(90)
    forward(side_length)
    left(90)
    forward(side_length)
    left(90)
```

```python
draw_square(100)
forward(100)
draw_square(100)
```
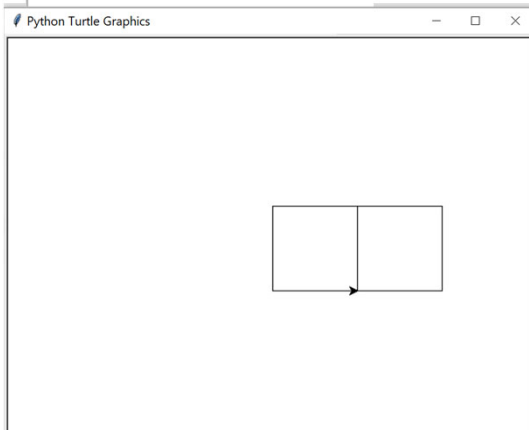


```python
draw_square(100)
left(180)
draw_square(100)
```

Namespaces

Namespace
- Builtin
- Global
- Enclosed
- Local
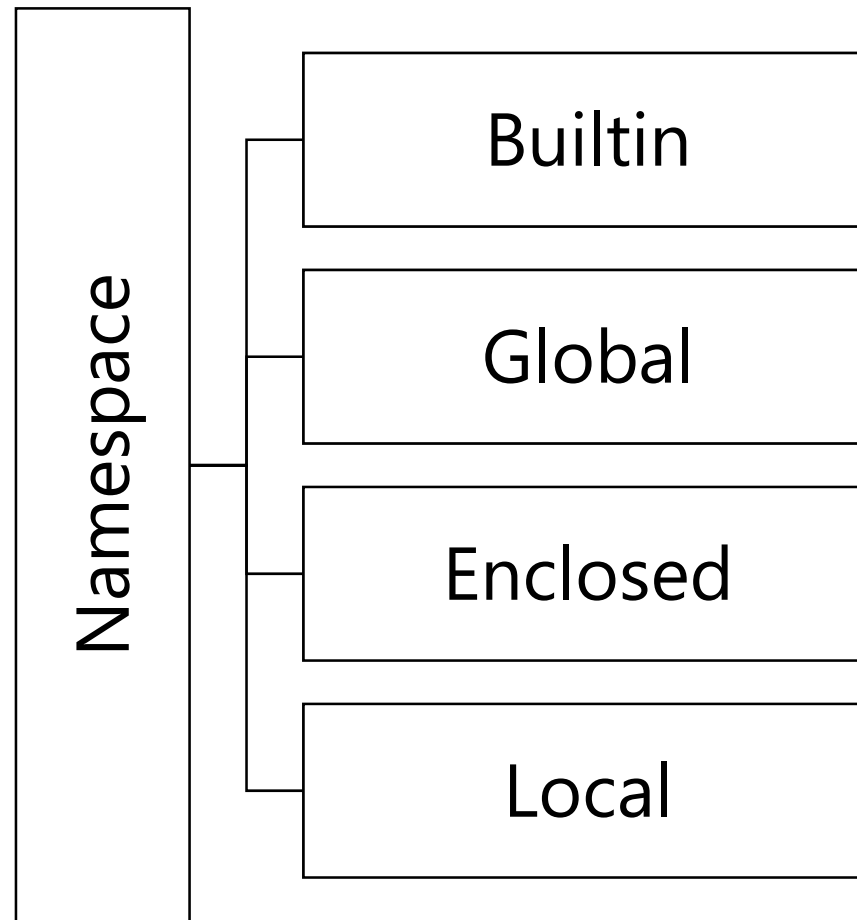
Namespaces: User-defined functions

Local namespace
is created when
the user-defined
function is called

Local Namespace

Global Namespace

Builtin Namespace
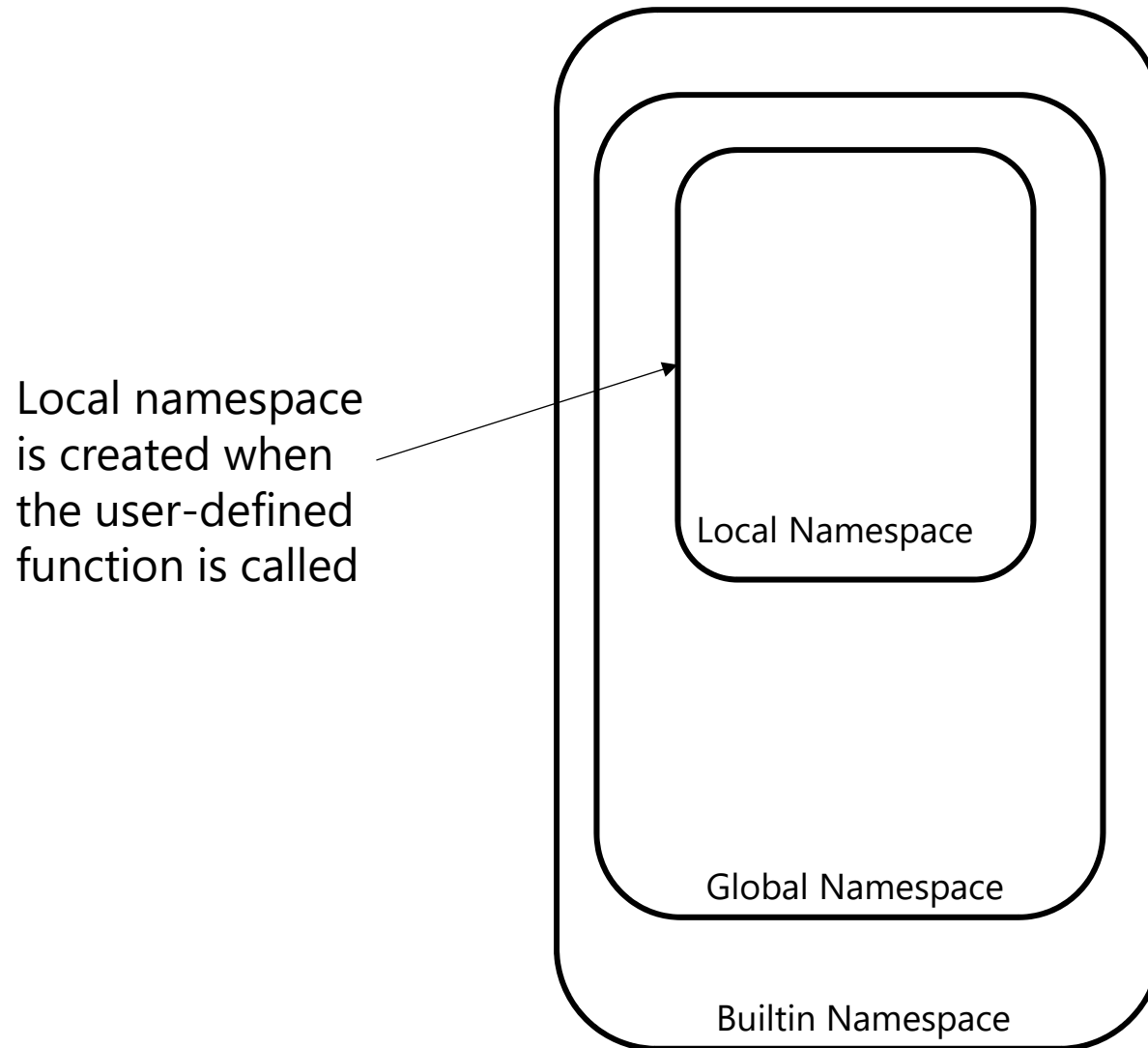
Namespaces: Imported Modules and User-defined functions

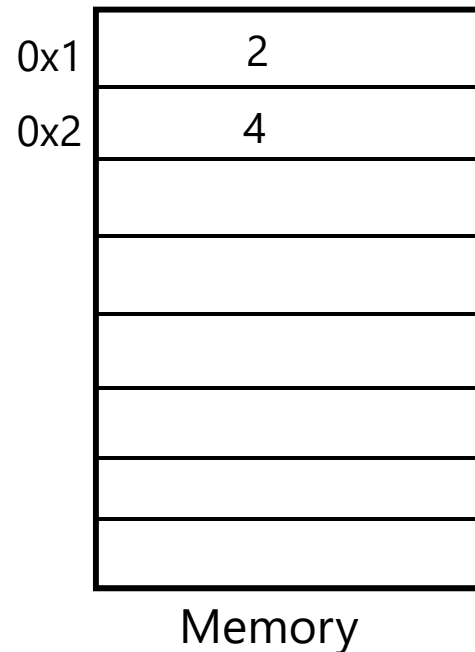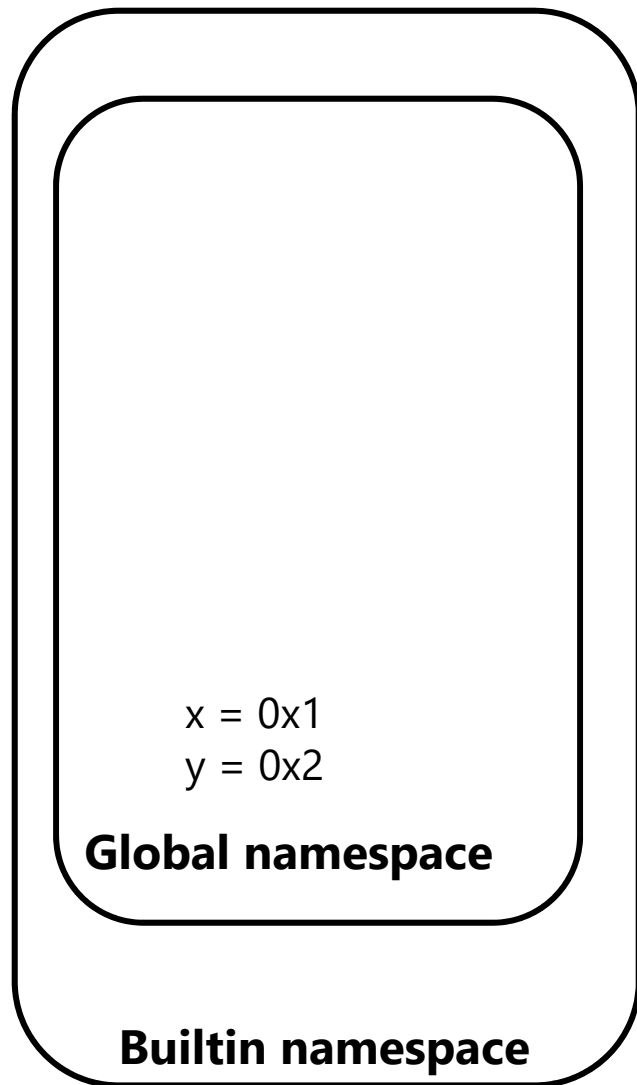Local namespace is created with in global namespace when the user-defined function is called

Dedicated namespace for each imported module

Local Namespace

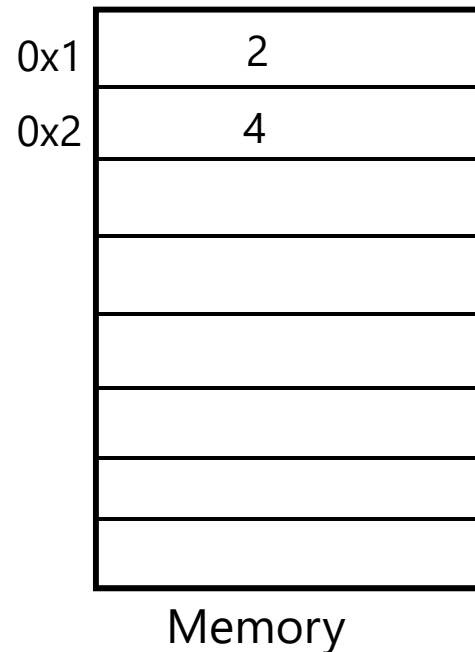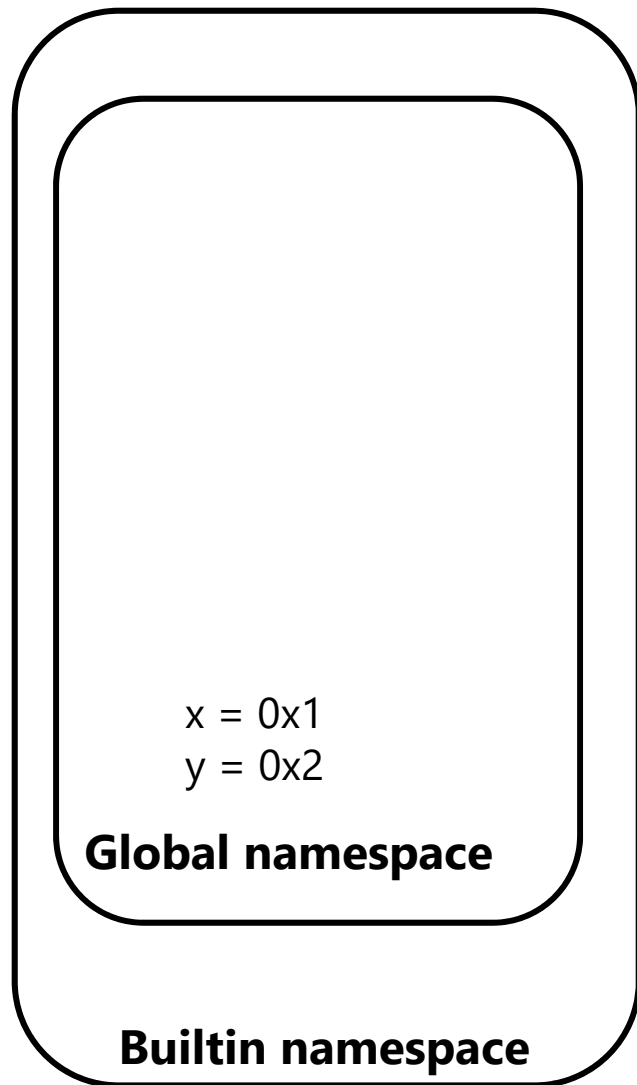imported_module_1 Namespace

imported_module_2 Namespace

Global Namespace

Builtin Namespace
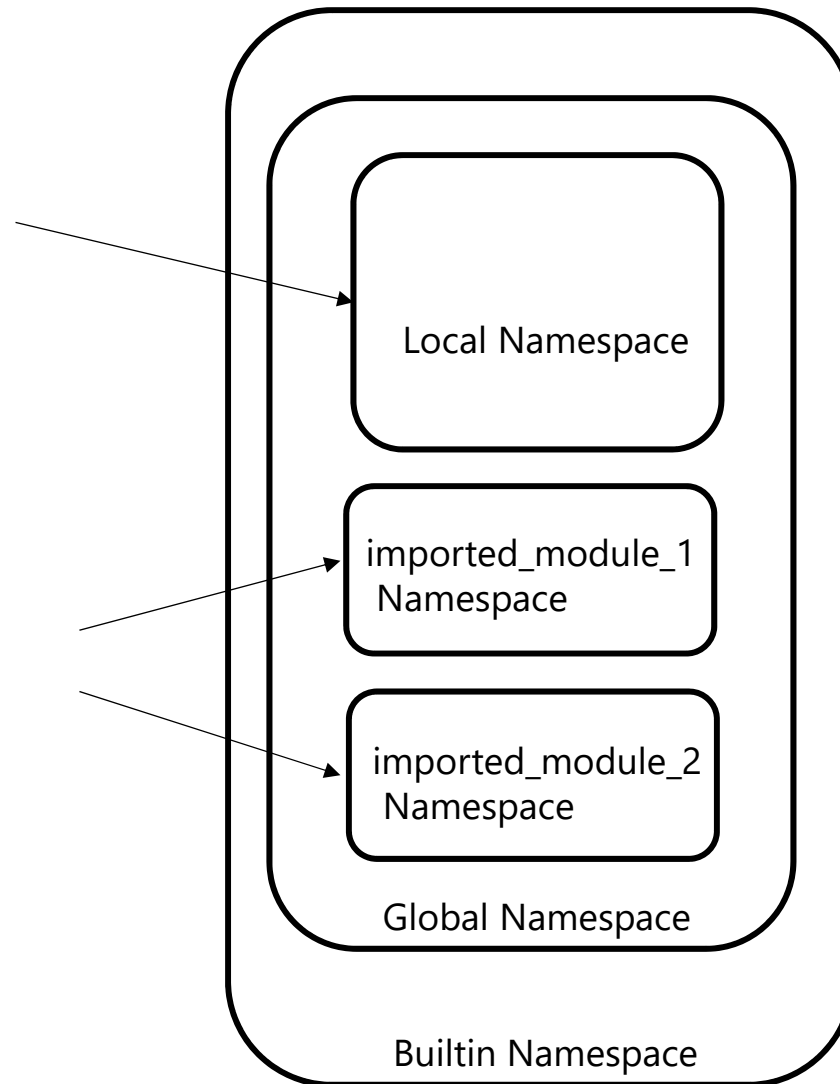
Functions: Tracing 1

```python
x = 2
y = 4
def sum_two_numbers(x,y):
    return x + y
z = sum_two_numbers(3,6)
print(z)
```
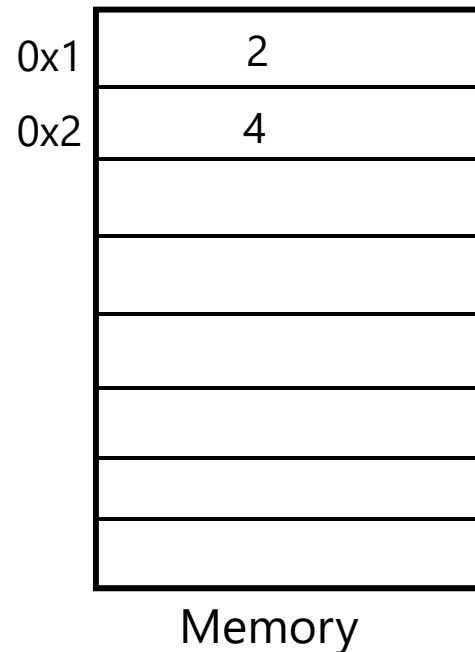
Functions: Tracing 1

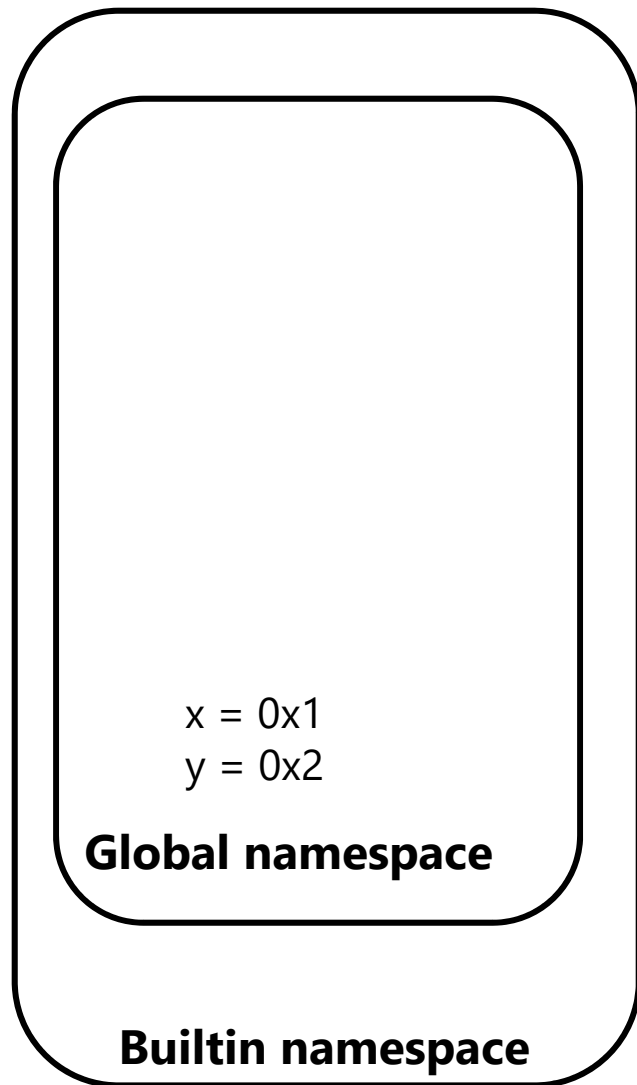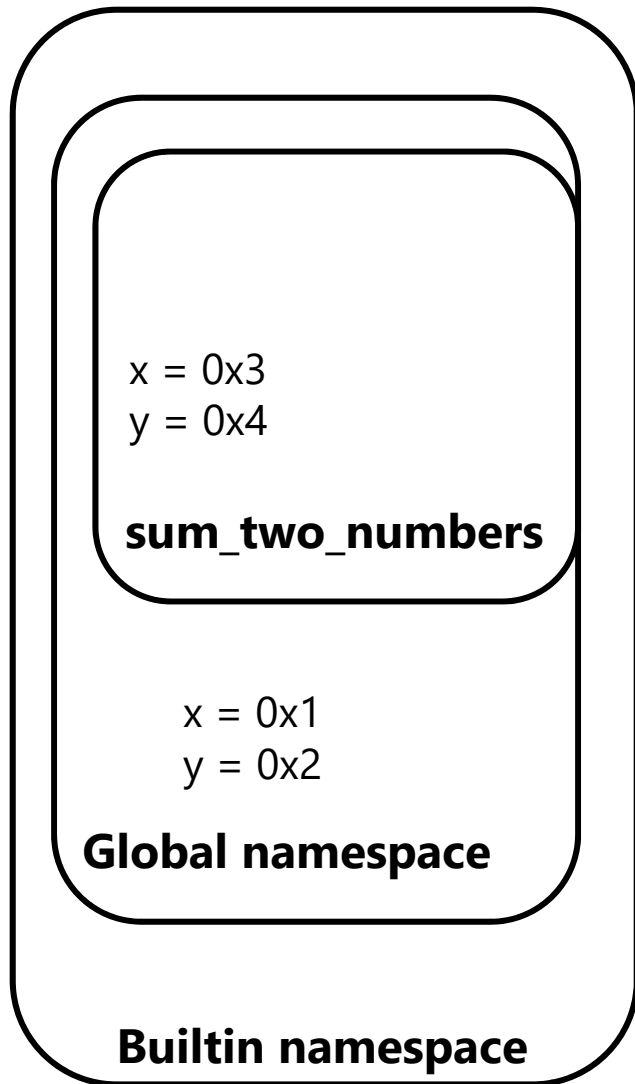|	
0x1	2
0x2	4

Memory

x = 0x1
y = 0x2

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```

Functions: Tracing 1

Memory table:

0x1	2
0x2	4

Memory

Global namespace

x = 0x1
y = 0x2

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```

Functions: Tracing 1

Memory

| 0x1 | 2 |
| 0x2 | 4 |

x = 0x1
y = 0x2

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```
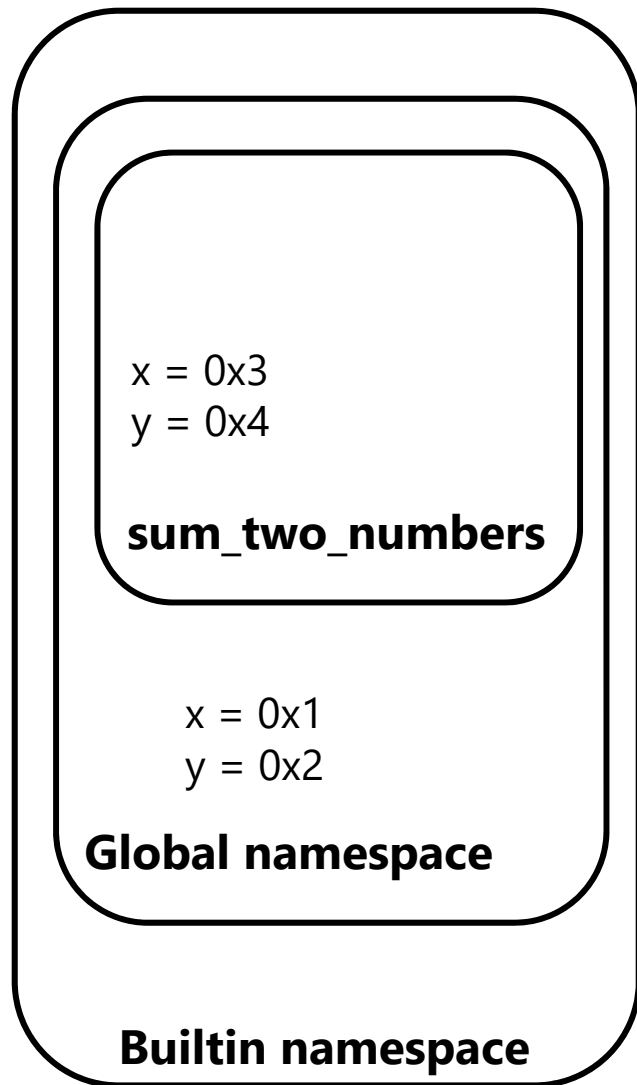
Functions: Tracing 1

0x1	2
0x2	4
0x3	3
0x4	6

Memory

x = 0x3
y = 0x4

sum_two_numbers

x = 0x1
y = 0x2

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```

Functions: Tracing 1

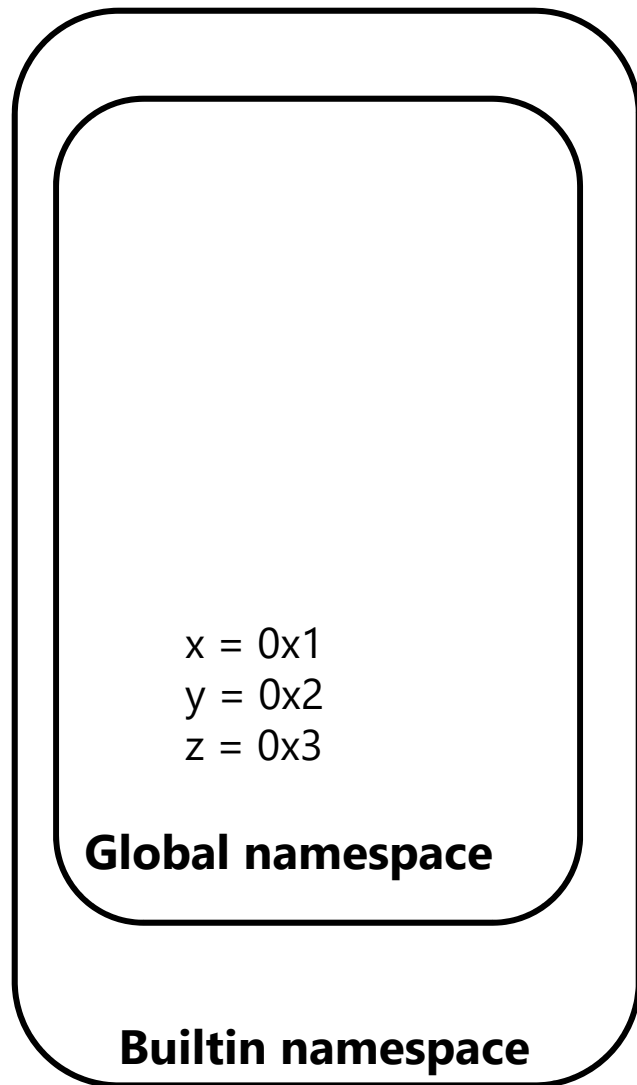|	
0x1	2
0x2	4
0x3	3
0x4	6

Memory

x = 0x3
y = 0x4

sum_two_numbers

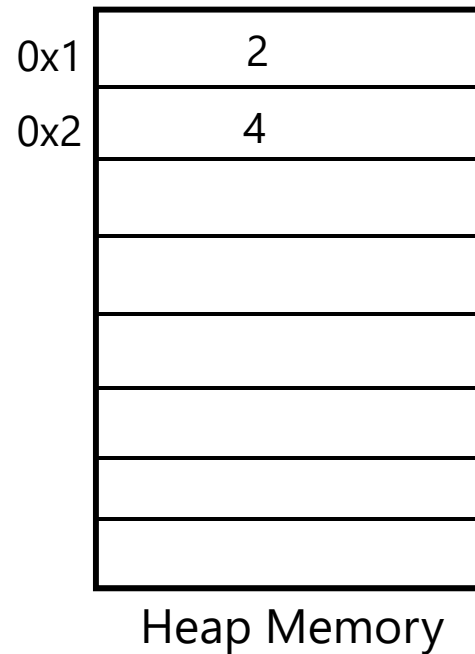x = 0x1
y = 0x2

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```

Functions: Tracing 1

Memory

0x1	2
0x2	4
0x3	9

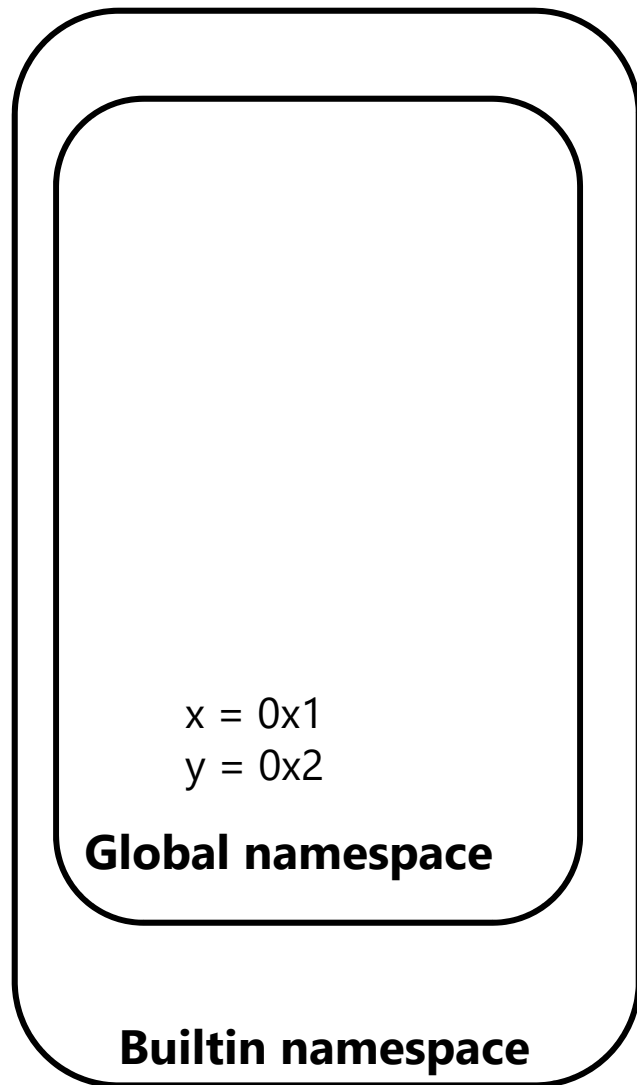Global namespace

x = 0x1
y = 0x2
z = 0x3

Builtin namespace

```
1  x = 2
2  y = 4
3  def sum_two_numbers(x,y):
4      return x + y
5  z = sum_two_numbers(3,6)
6  print(z)
7
```
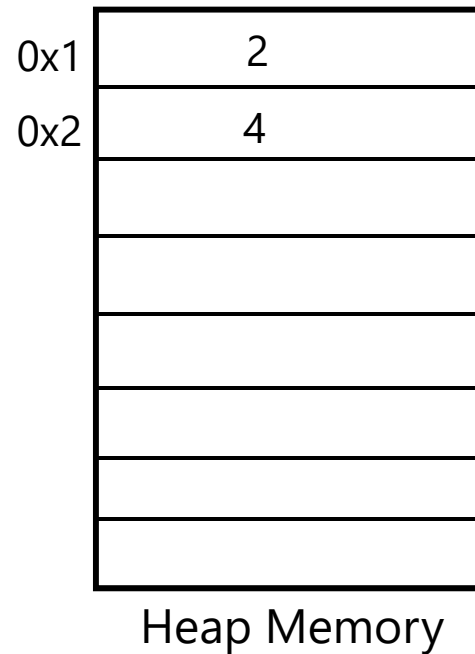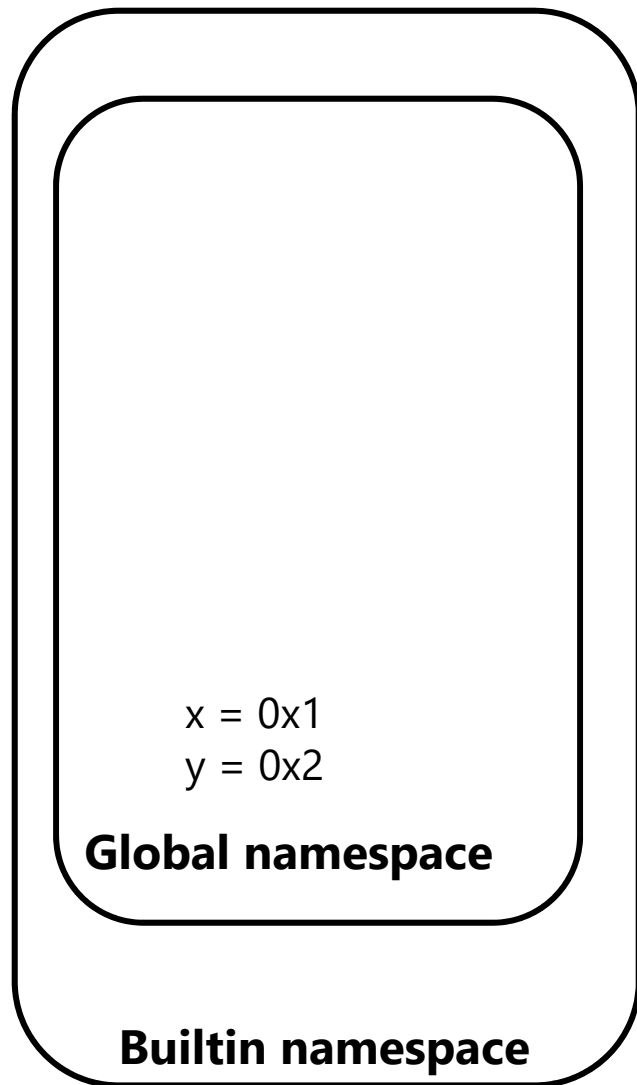
Functions: Tracing 2

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
8
```

Functions: Tracing 2

0x1	2
0x2	4

Heap Memory

x = 0x1
y = 0x2

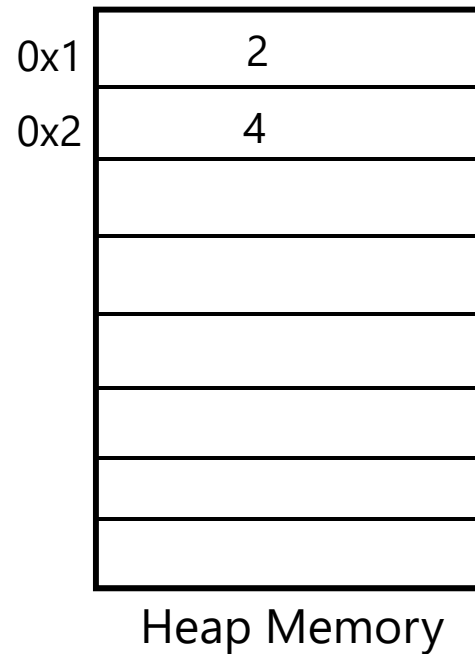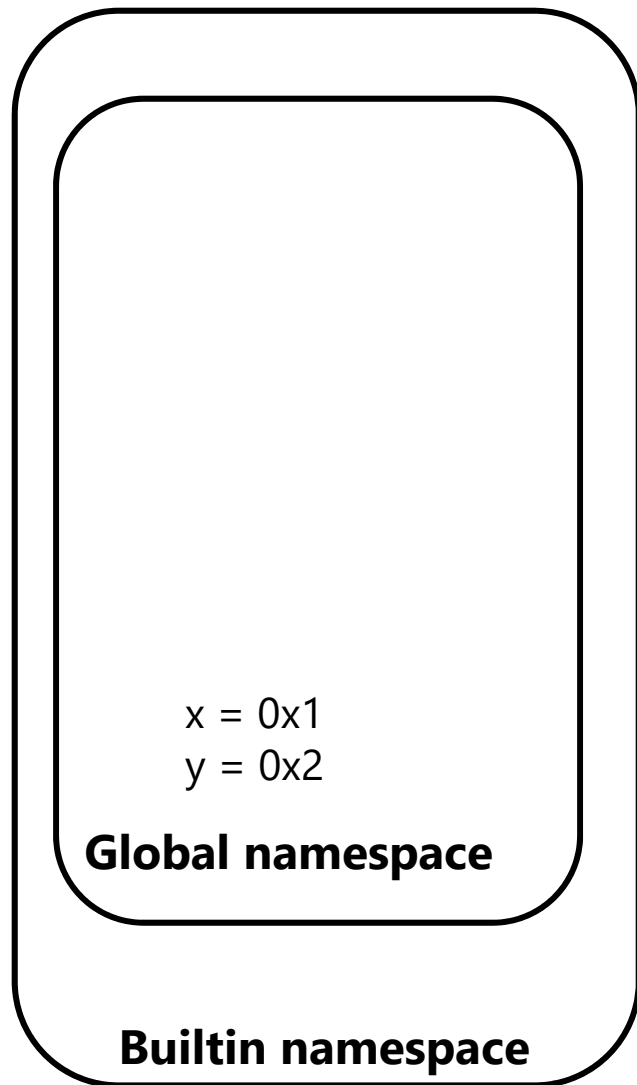**Global namespace**

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Functions: Tracing 2

Heap Memory

0x1	2
0x2	4

x = 0x1
y = 0x2

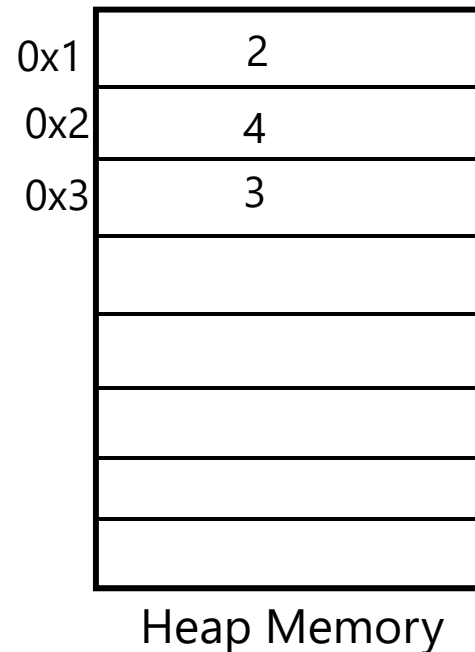**Global namespace**

Builtin namespace

```
1  x = 2
2  y = 4
3  def sum_two_numbers(x):
4      return x+y
5
6  z = sum_two_numbers(3)
7  print(z)
8
```

Functions: Tracing 2

0x1	2
0x2	4

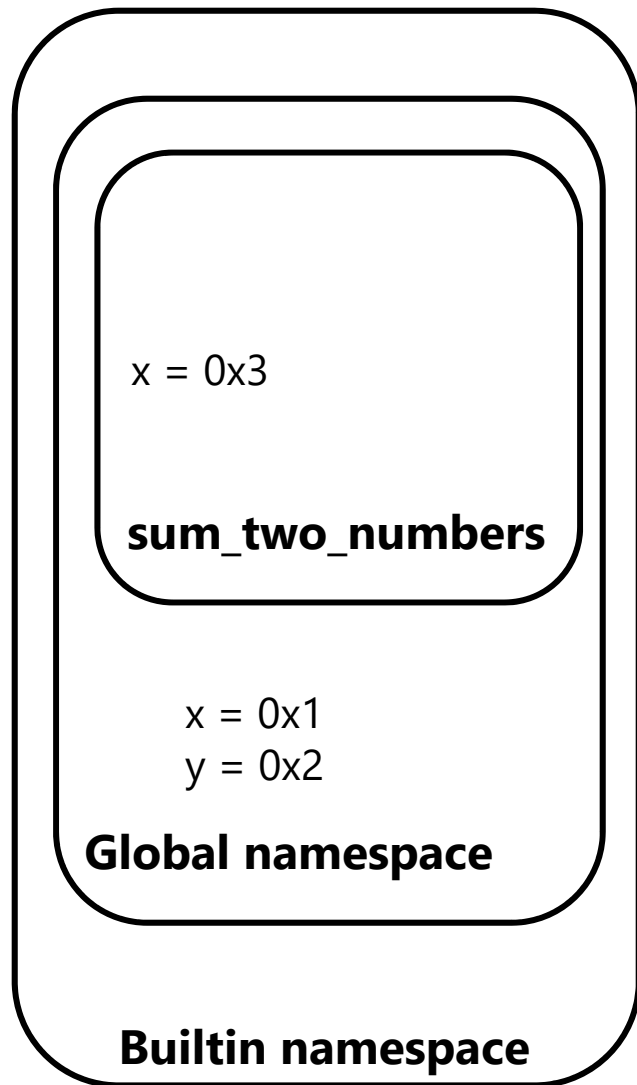Heap Memory

x = 0x1
y = 0x2

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
8
```

Functions: Tracing 2

Heap Memory:

Address	Value
0x1	2
0x2	4
0x3	3

Namespaces:
- sum_two_numbers: x = 0x3
- Global namespace: x = 0x1, y = 0x2
- Builtin namespace

```
1  x = 2
2  y = 4
3  def sum_two_numbers(x):
4      return x+y
5
6  z = sum_two_numbers(3)
7  print(z)
```
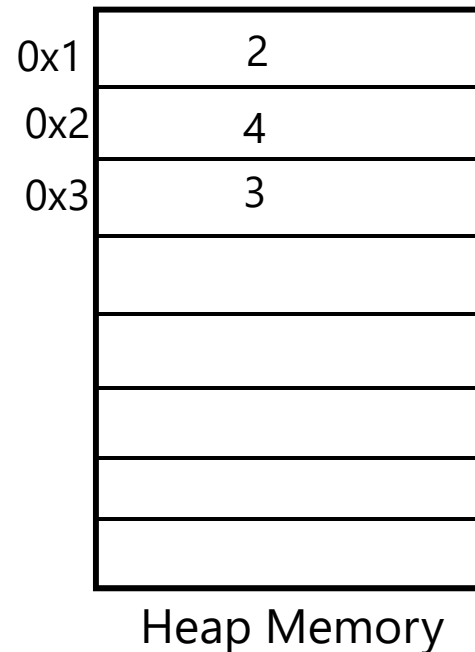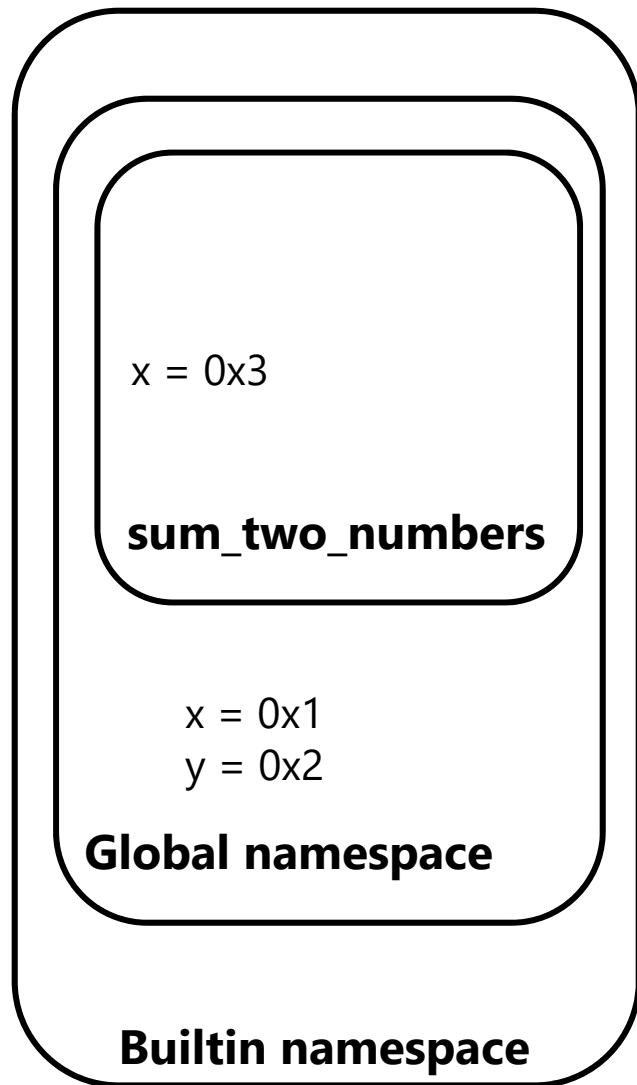
Functions: Tracing 2

Heap Memory

0x1	2
0x2	4
0x3	3

x = 0x3

sum_two_numbers
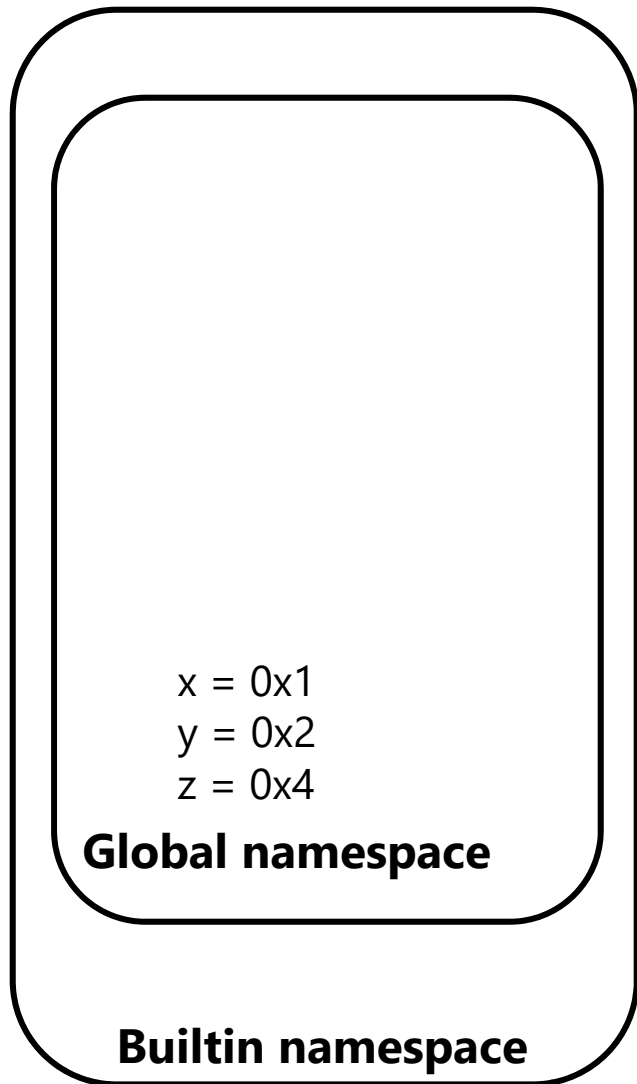
x = 0x1
y = 0x2

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Functions: Tracing 2

0x1	2
0x2	4
0x3	3
0x4	7

Heap Memory

x = 0x1
y = 0x2
z = 0x4

Global namespace

Builtin namespace

```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Side-effects

- Functions with side-effects
  - ➤ Side-effects: I/O tasks
    - ○ Taking inputs from keyboard, reading data from a file, etc.
    - ○ Printing output to screen, writing data to a file, etc.

Domain   *f*   Range

side-effect   I/O etc.

- Pure functions
  - ➤ Functions without side-effects
  - ➤ Only mapping
  - ➤ Outputs depend only on inputs
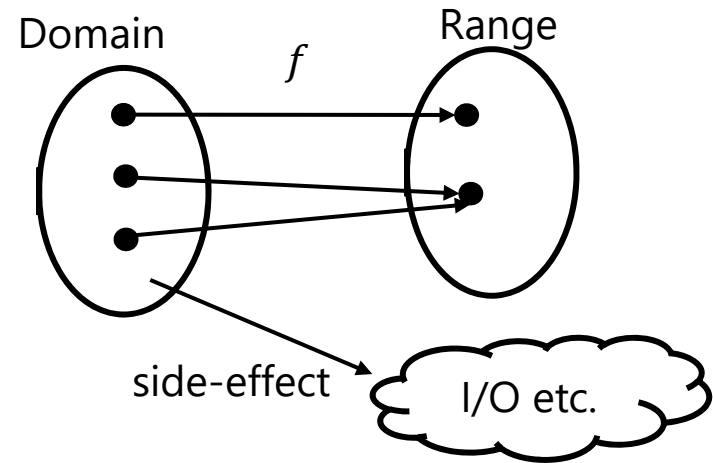
Pure Function

Pure function: Yes/No?

```
x = 2
y = 4
def sum_two_numbers(x,y):
    return x + y
z = sum_two_numbers(3,6)
print(z)
```

Pure function: Yes/No?

```
x = 2
y = 4
def sum_two_numbers(x):
    return x+y

z = sum_two_numbers(3)
print(z)
```

Summary

- Modules and Packages

- User-defined functions

- Argument Types

- Print vs Return

- Pure Function