IT5001 Software Development Fundamentals

2. Functions (Callable Units)

Rajendra Prasad Sirigina

August-2022

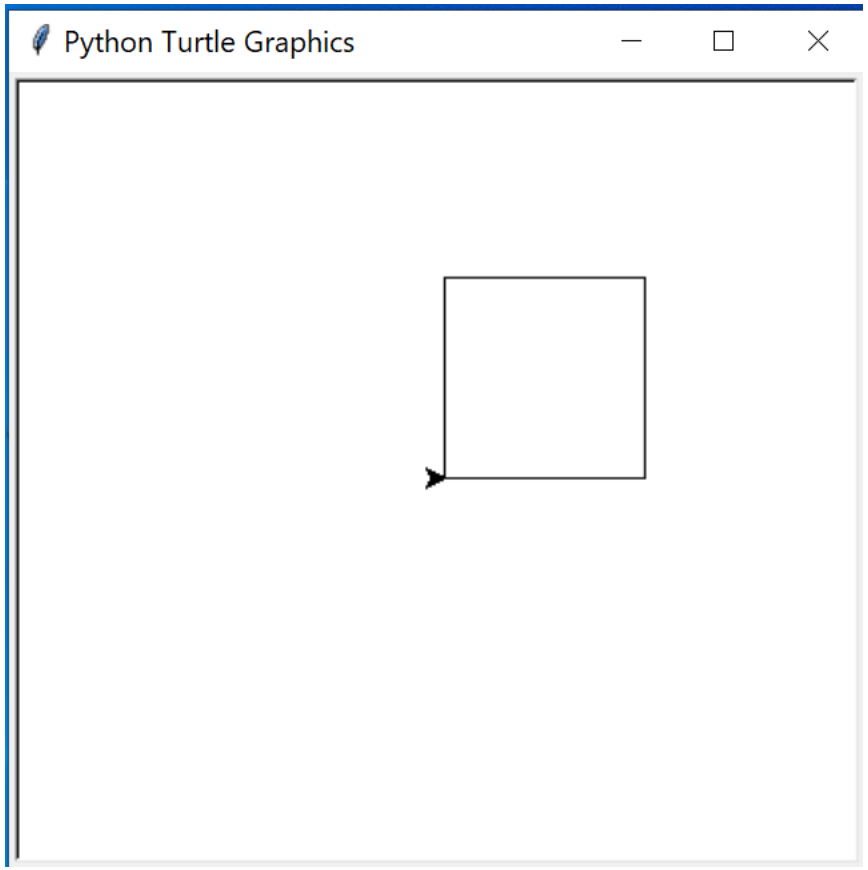
Agenda: Functions

- Python Modules and Packages
  - Namespaces
  - Scope
- User-defined functions
  - Arguments
    - Positional
    - Keyword
    - Default
  - Return Statement
    - print vs return
  - Function Tracing
    - Namespaces
    - Local namespace
- Pure Functions

Why Functions?

Example: Drawing a square

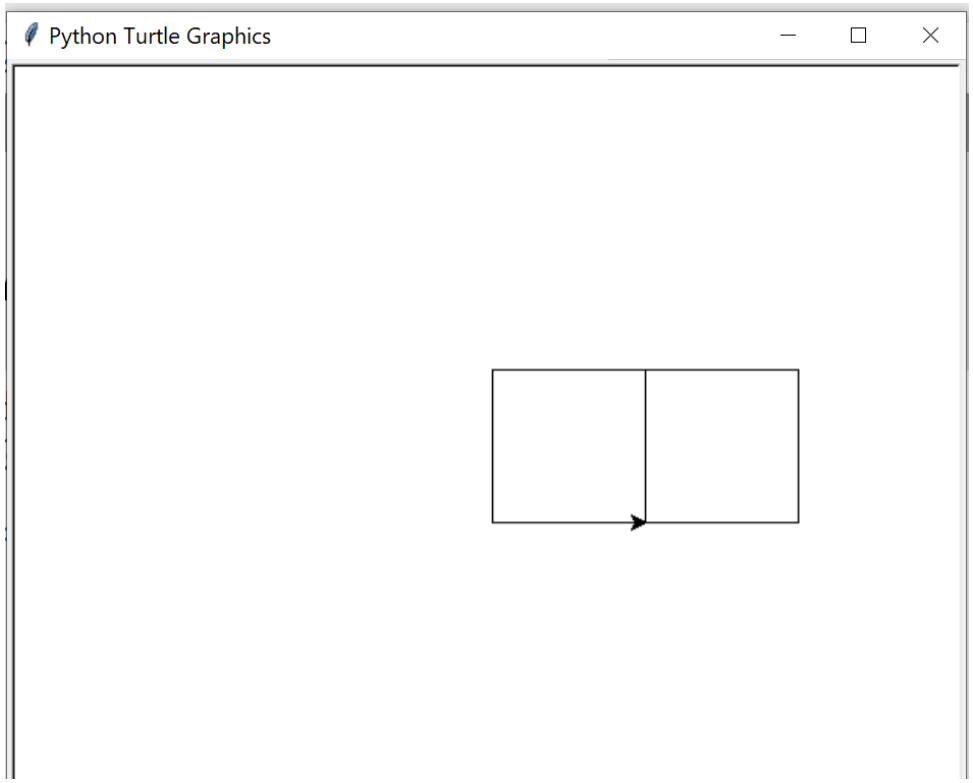
```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```



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

Drawing another square

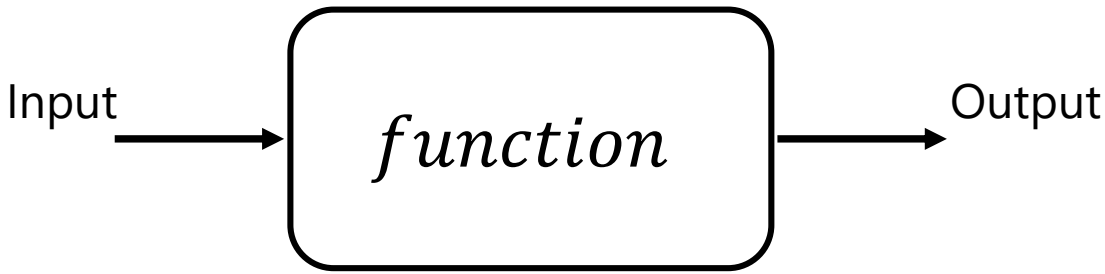
```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```



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



- Usability
  - User only needs to know input arguments and output data types

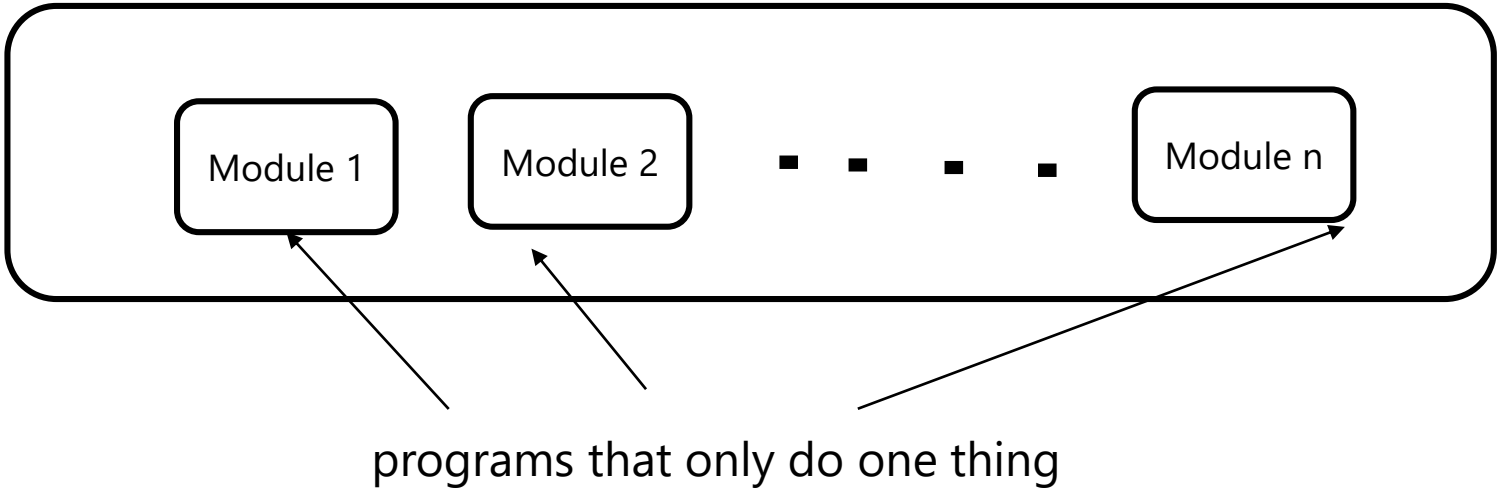
- Declarative style of programming
  - Tell the system what you want

Modularity

A Giant Module
(A program that do many things)

Vs

Modular Architecture



Functions: Mathematics

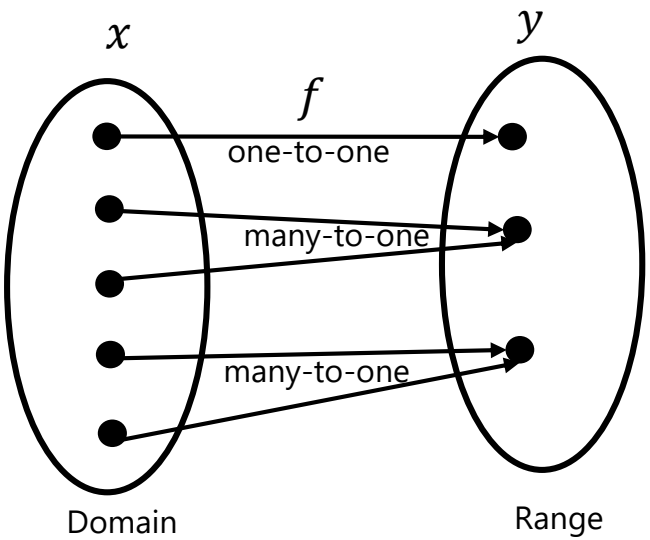
- A mapping from input to output

name of the function

$$y = f(x)$$

output

argument

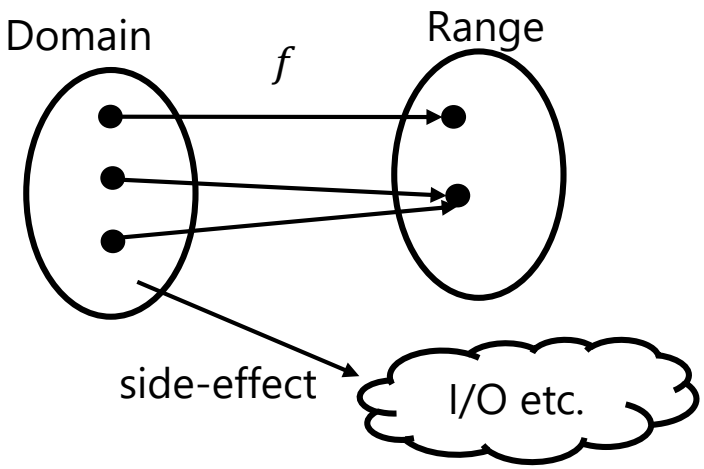


Functions in Programming

- 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
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

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

Functions from Python Modules and Packages

- Import packages
  - Example:
    - math package

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

- 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 *

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

Namespaces: Importing Modules

Syntax:

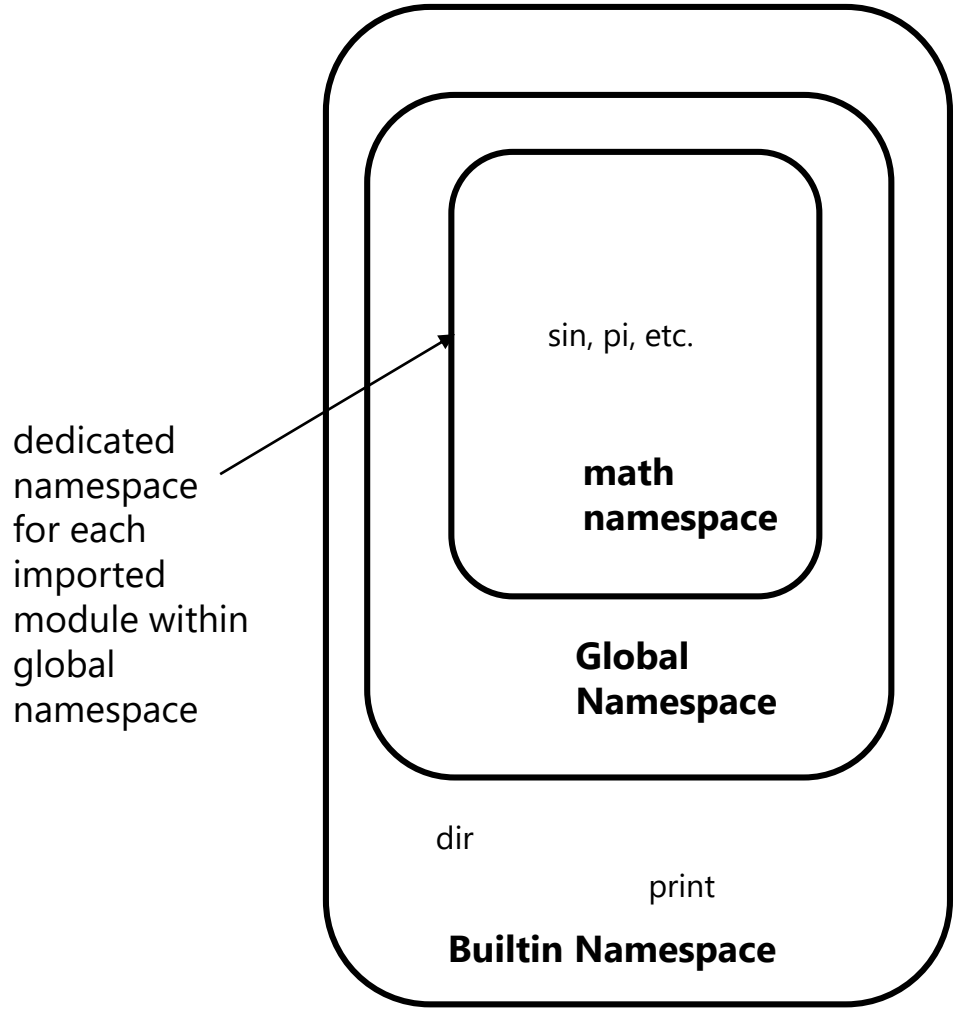
import *<module_name>*

<module_name>.<object_name>

Example:

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

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



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', 'expm1', '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
u', 'trunc']
```

Namespaces: Importing Multiple Modules

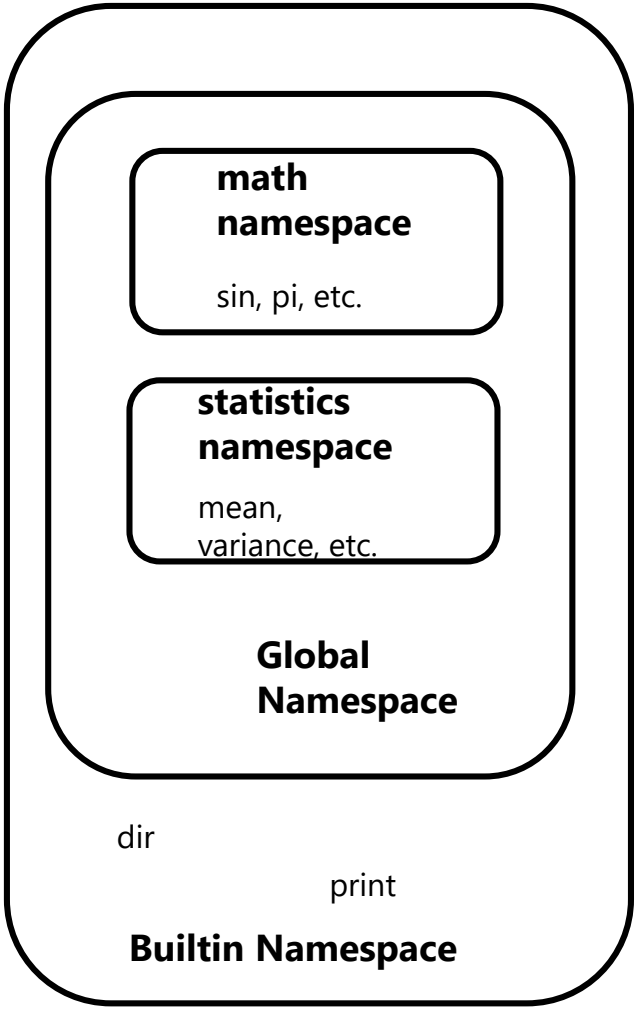
Syntax:

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

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

Example:

```
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_}')
```



Namespaces: Importing Objects from Modules

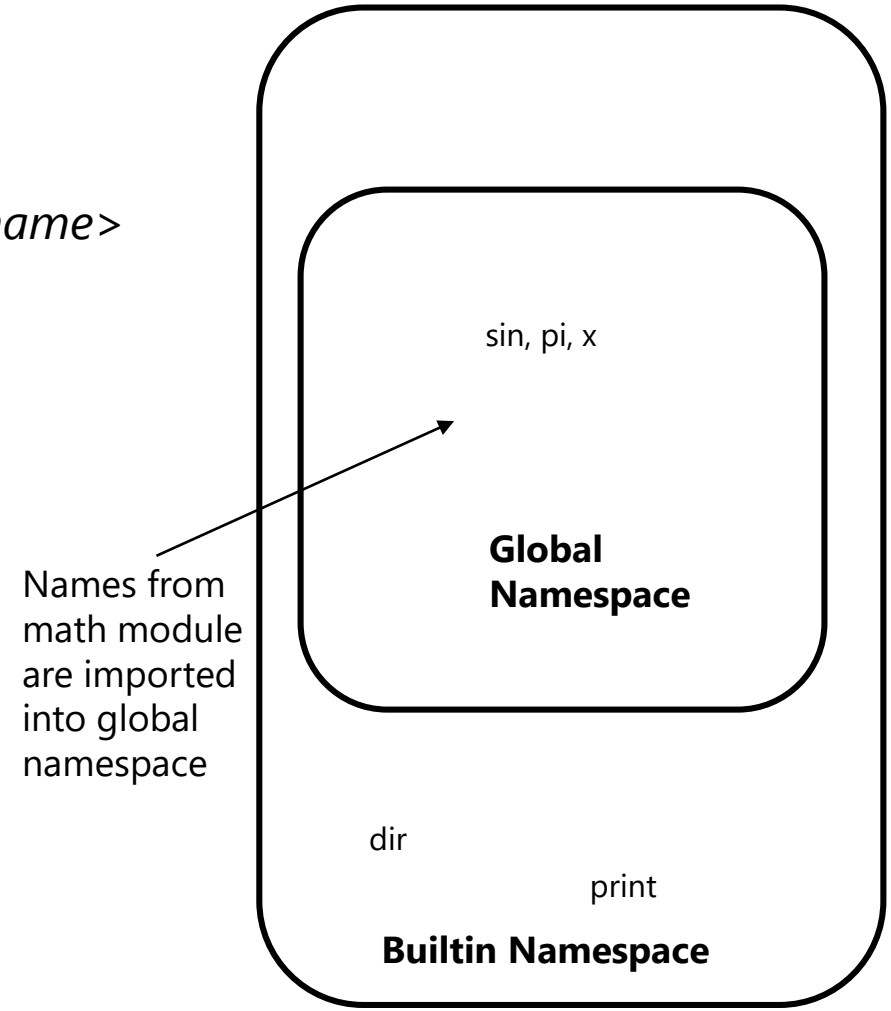
Syntax:

from *<module_name>* import *<object_name>*

<module_name>.<object_name>

Example:

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



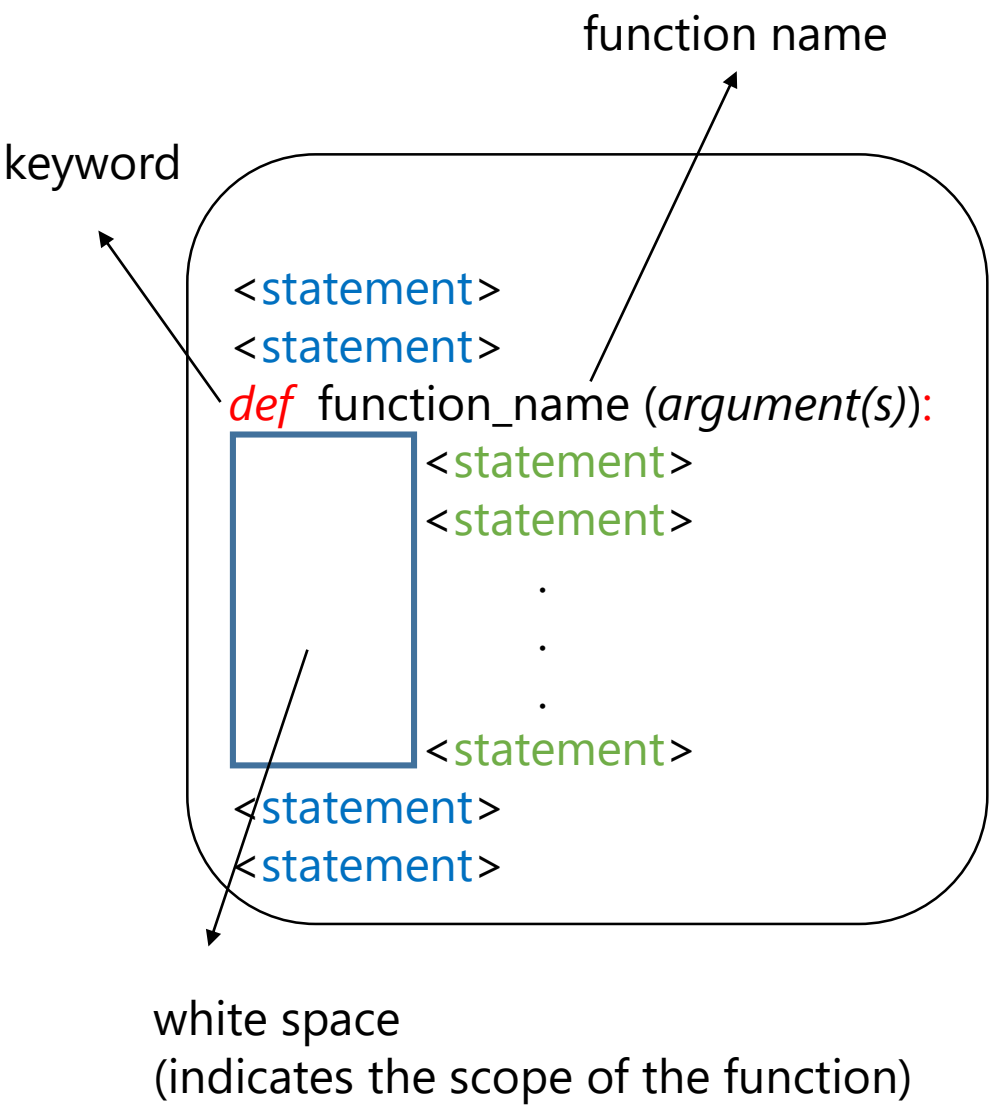
You don’t want to be like this in real life

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



y = function_name(*argument*)

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

<statement>: Statement that is out of function scope

User-Defined Functions

```
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

```
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

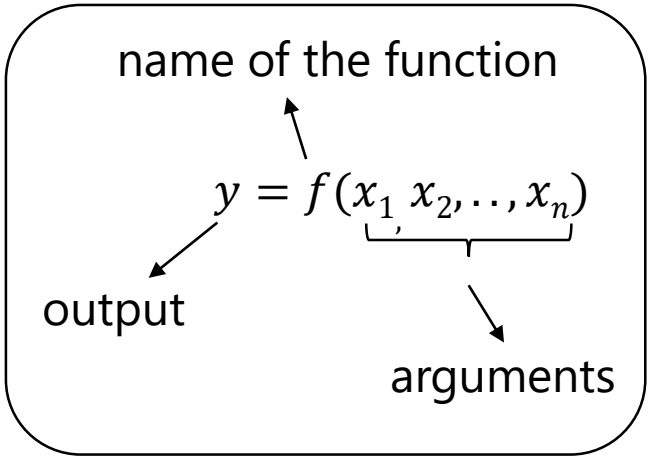
white space

Value returned by the function

Arguments

- Provide input to functions

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




```
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

```
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

```
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

```
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

```
>>> 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

```
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

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

- Default argument can be assigned a different value

```
>>> 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?

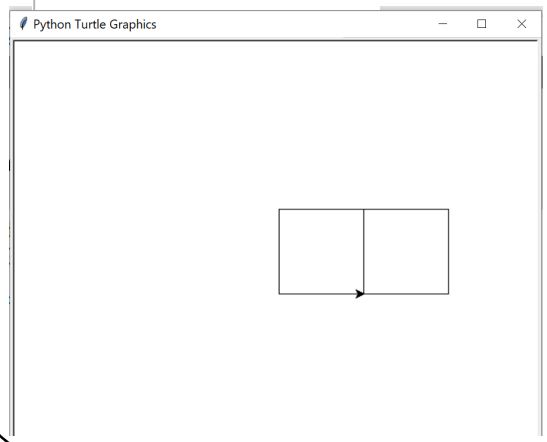
```
from turtle import *
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```

- Retain similarities
- Parametrize differences

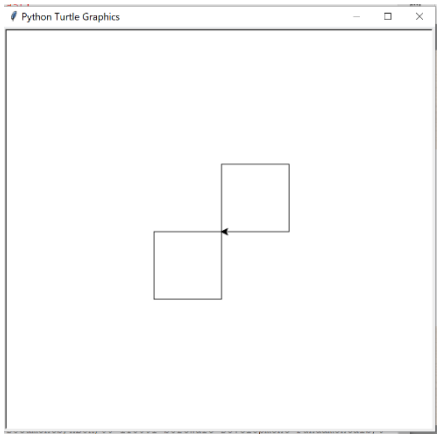
Drawing Squares using Functions

```
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)
```

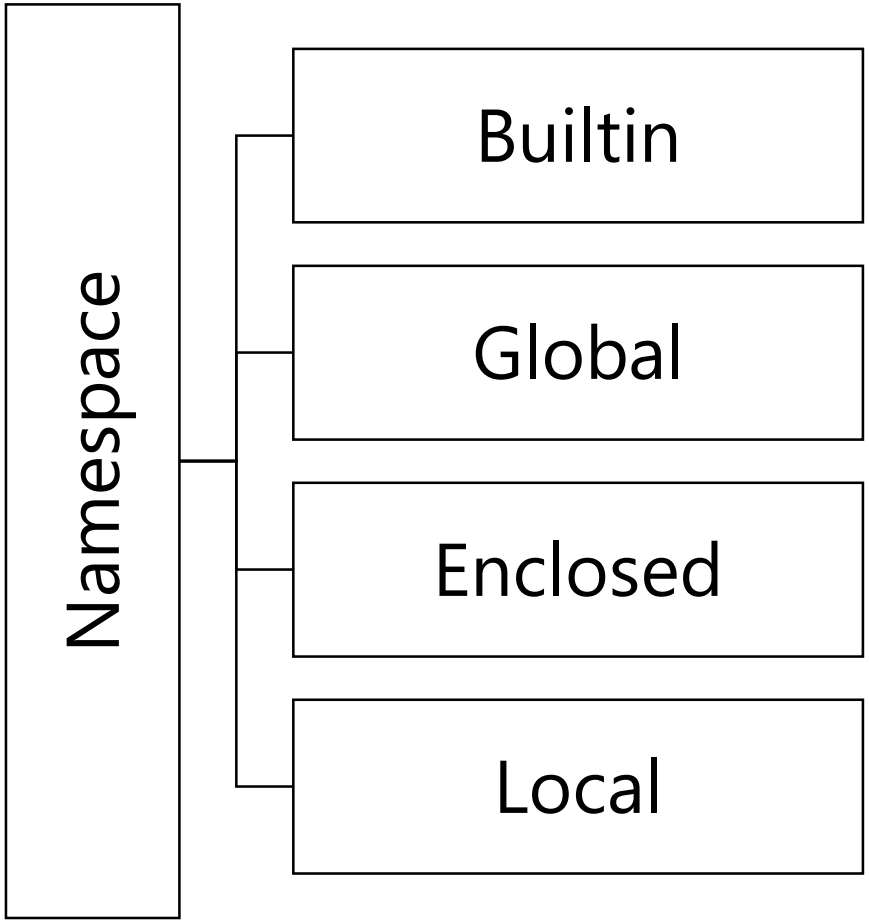
```
draw_square(100)
forward(100)
draw_square(100)
```




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

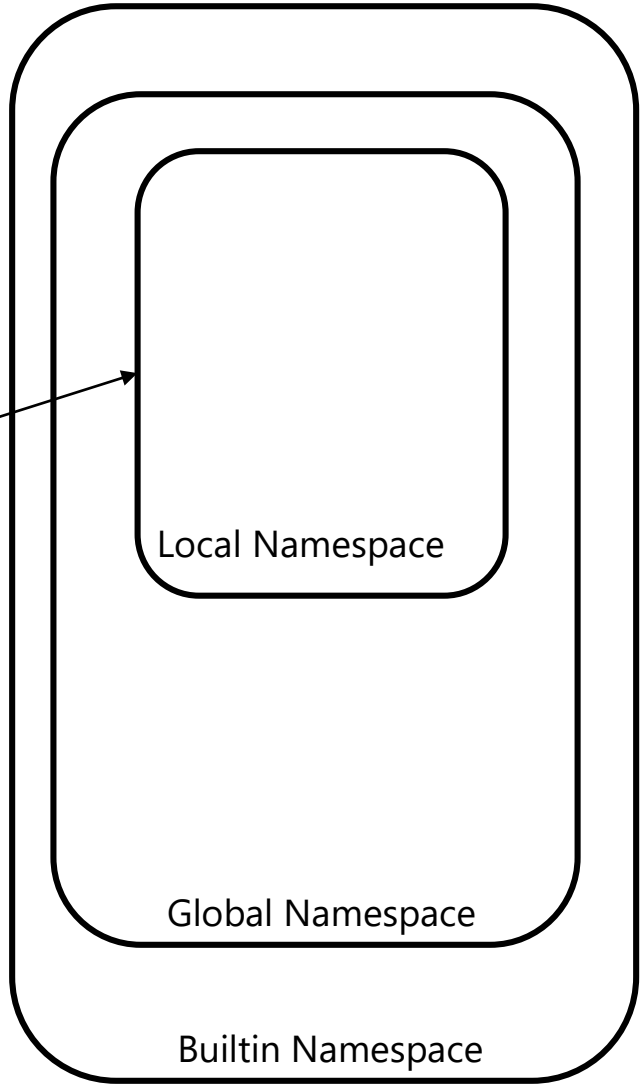


Namespaces



Namespaces: User-defined functions

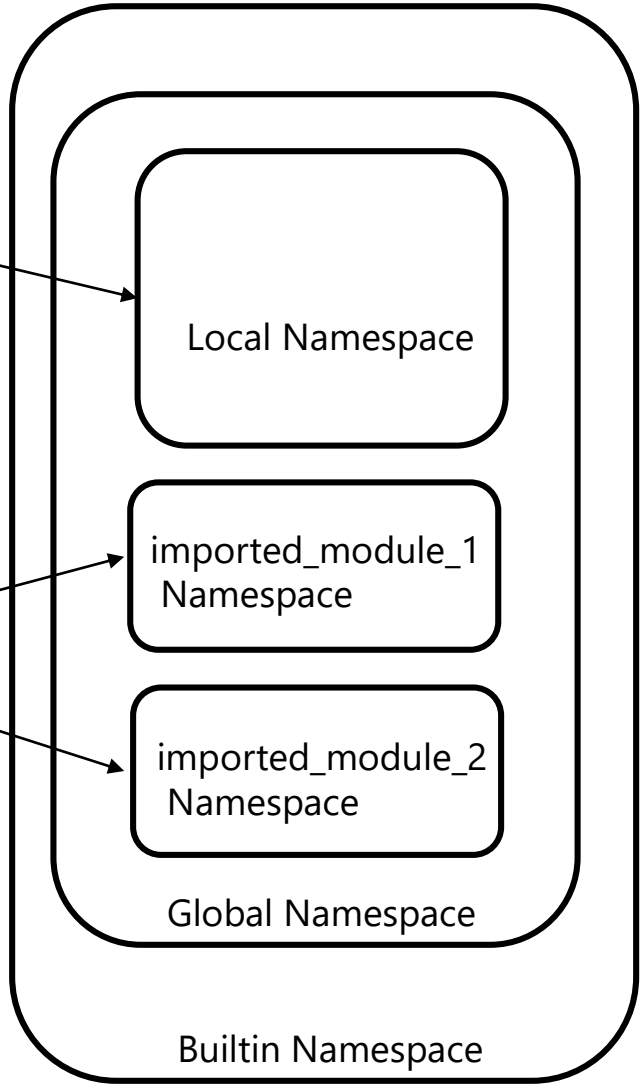
Local namespace is created when the user-defined function is called



Namespaces: Imported Modules and User-defined functions

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

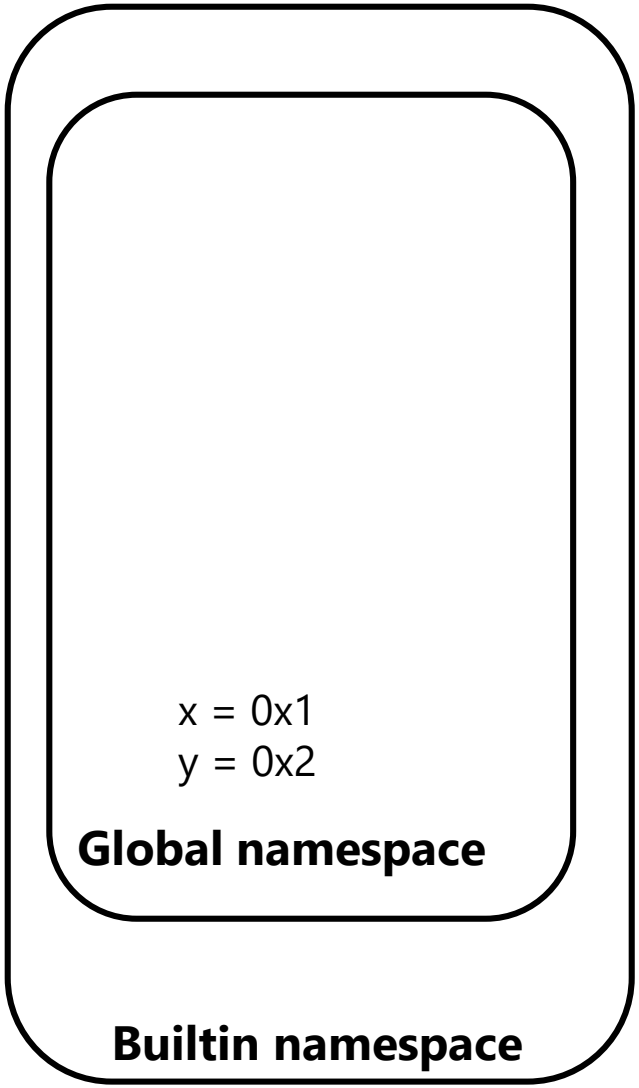
Dedicated namespace for each imported module

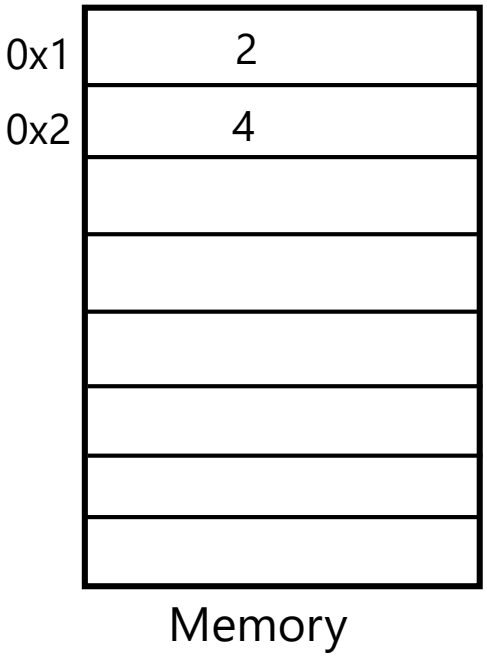


Functions: Tracing 1

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

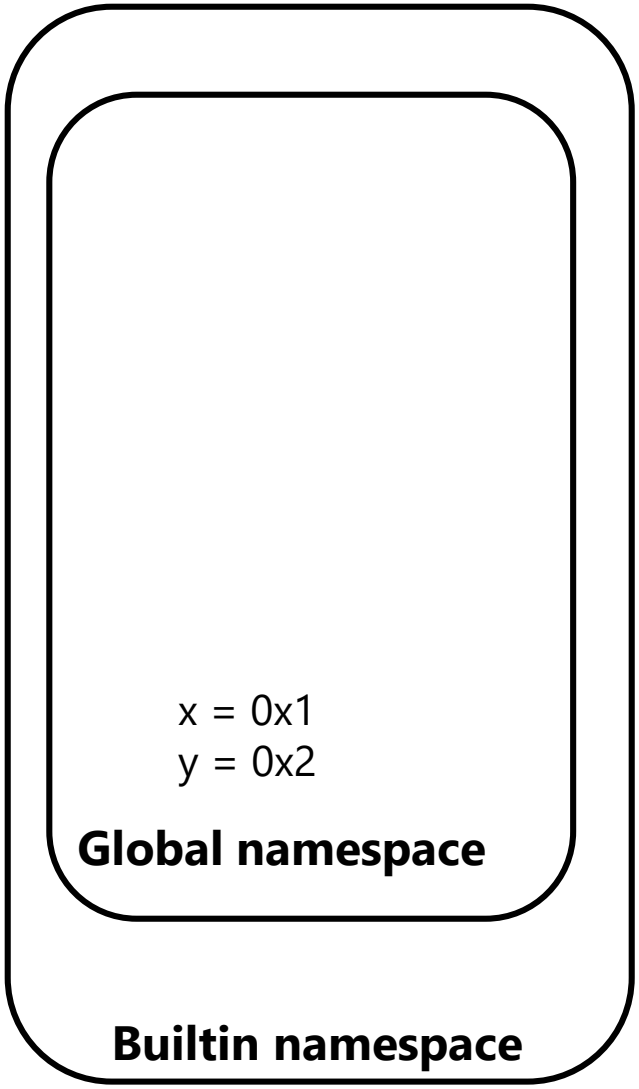
Functions: Tracing 1

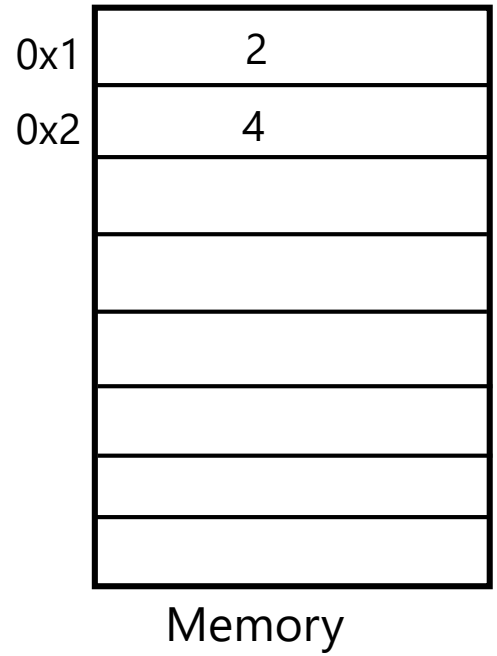





```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```

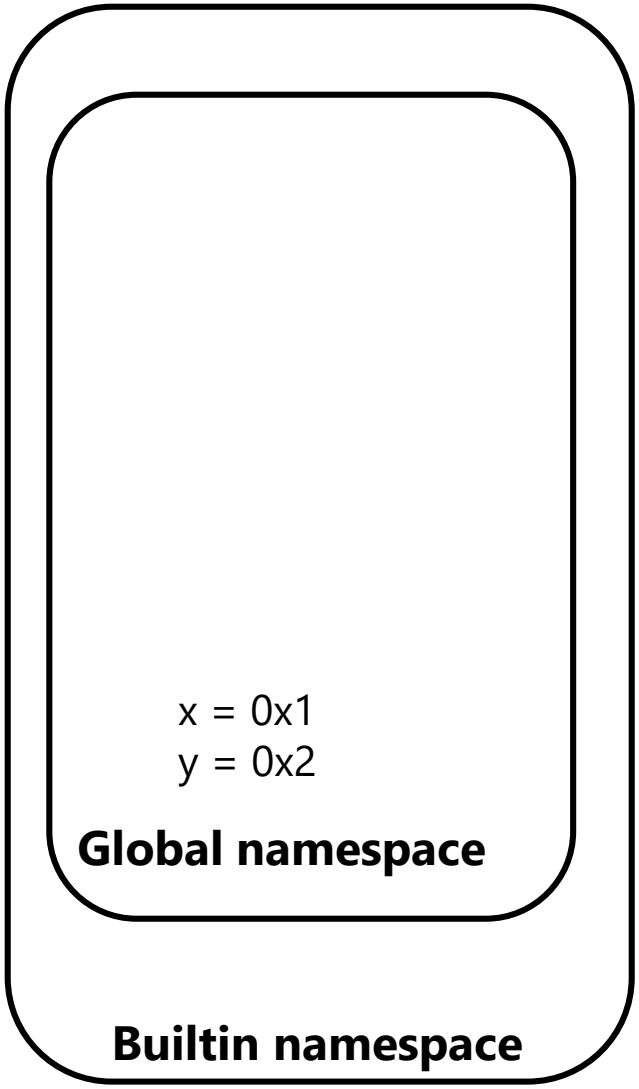
Functions: Tracing 1

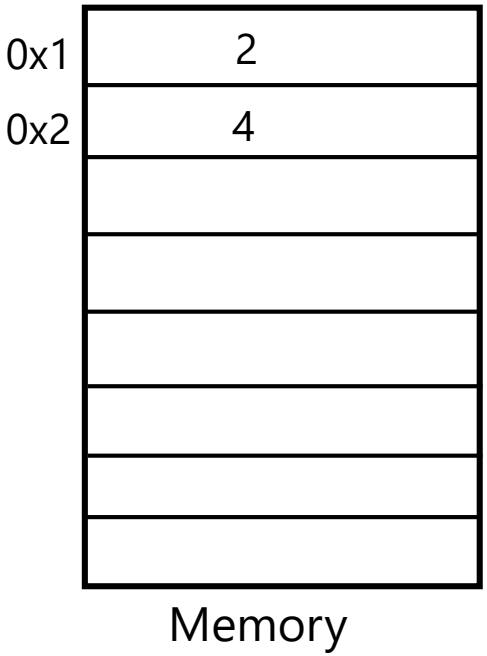





```
x = 2
y = 4
def sum_two_numbers(x,y):
    return x + y
z = sum_two_numbers(3,6)
print(z)
```

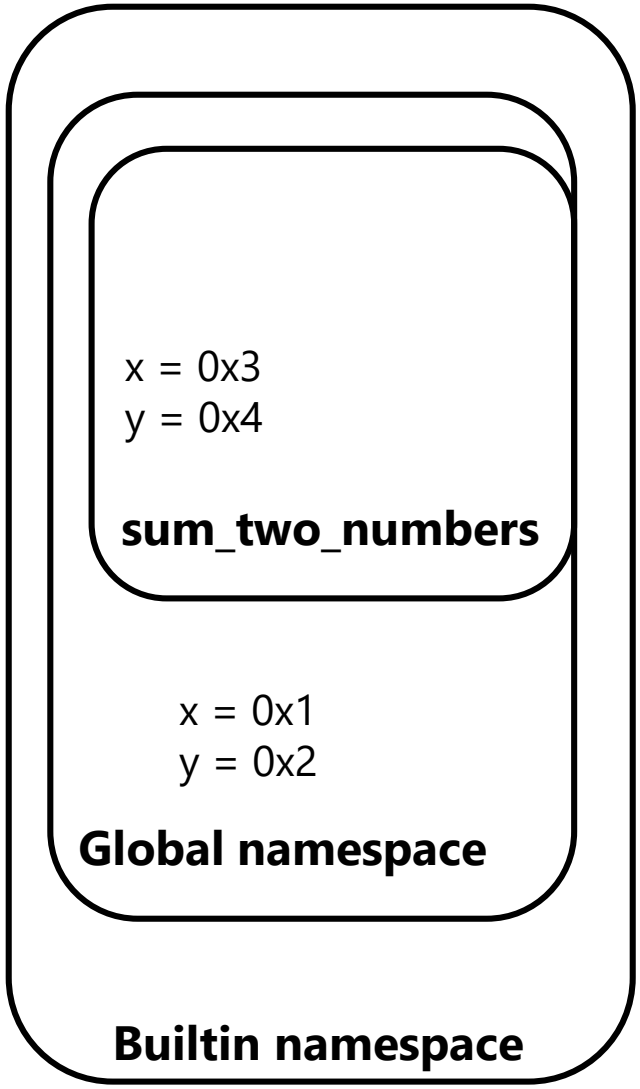
Functions: Tracing 1

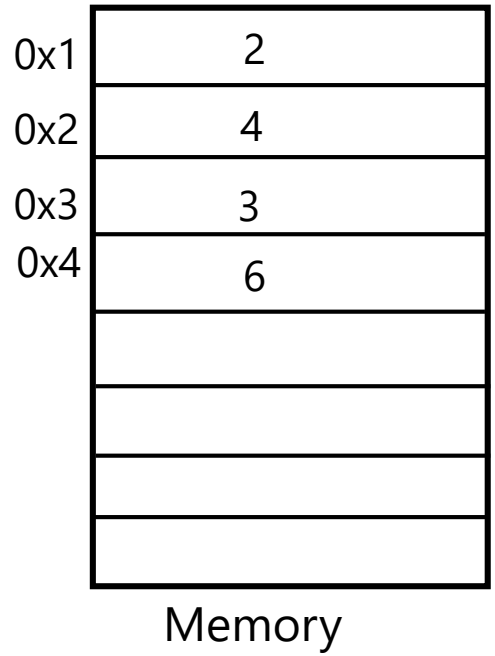





```
x = 2
y = 4
def sum_two_numbers(x,y):
    return x + y
z = sum_two_numbers(3,6)
print(z)
```

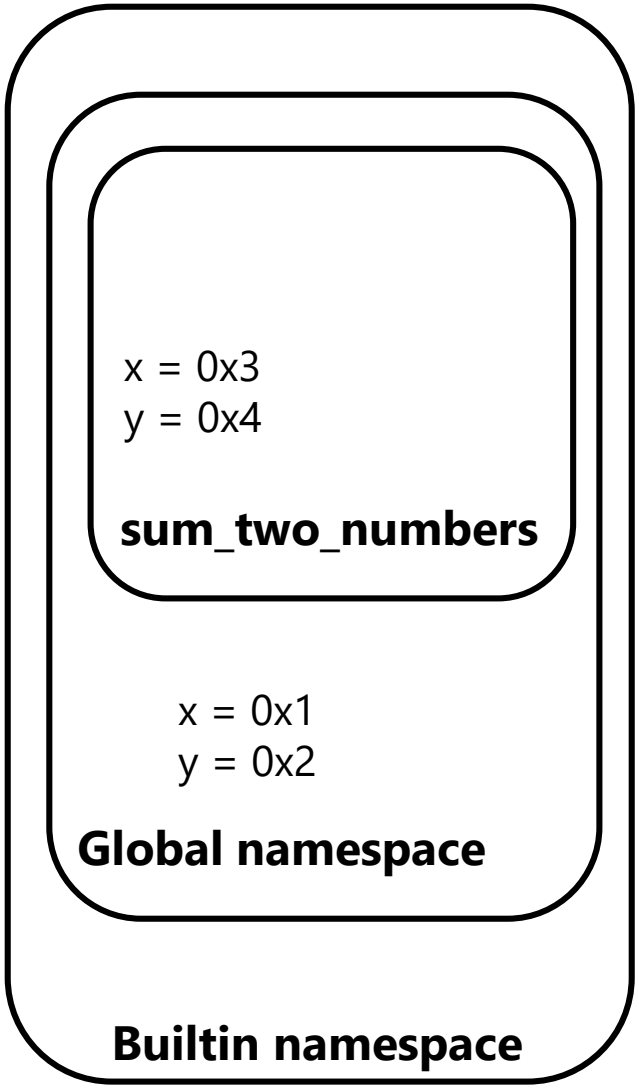
Functions: Tracing 1

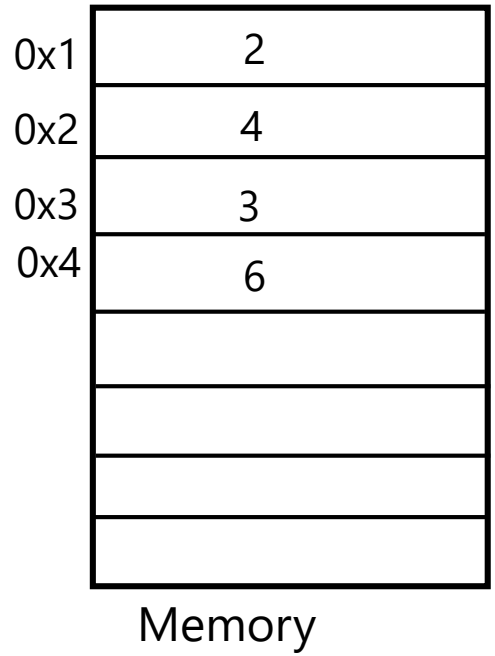





```
x = 2
y = 4
def sum_two_numbers(x,y):
    return x + y
z = sum_two_numbers(3,6)
print(z)
```

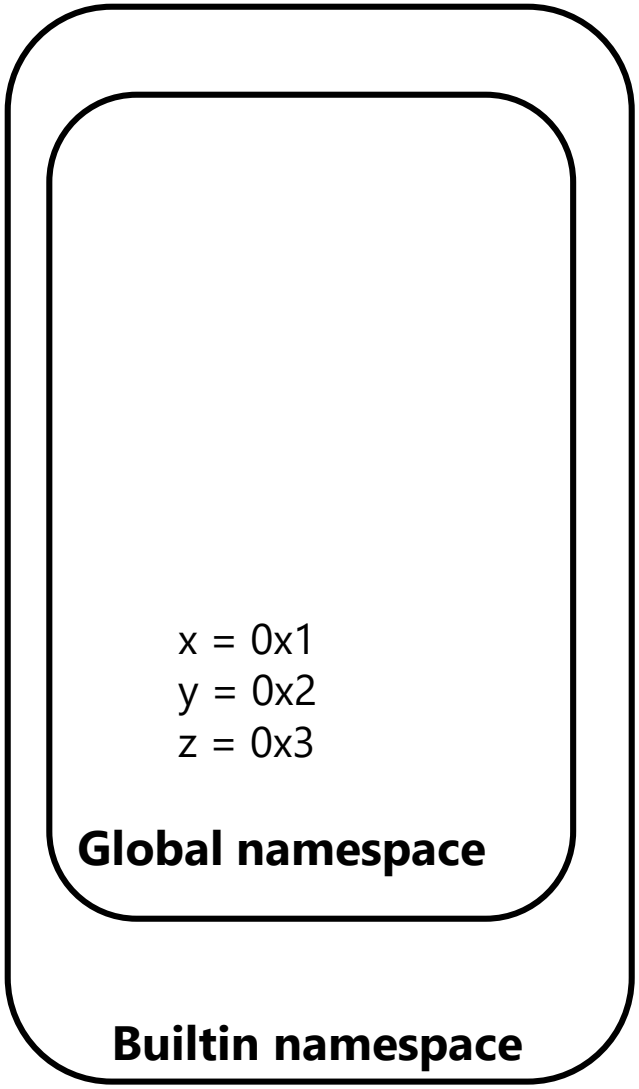
Functions: Tracing 1

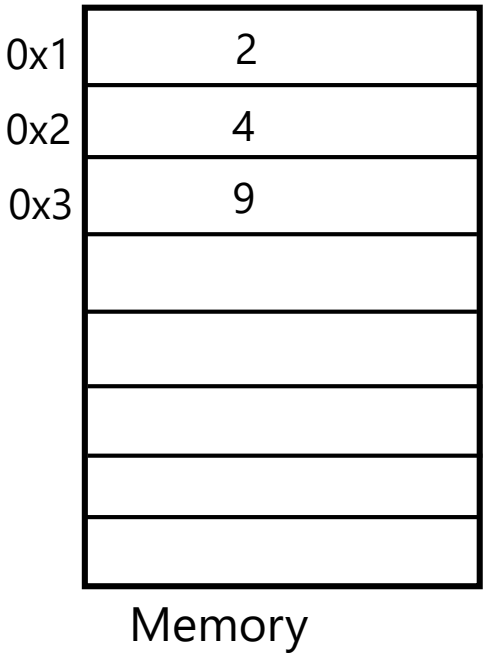





```
1 x = 2
2 y = 4
3 def sum_two_numbers(x,y):
4     return x + y
5 z = sum_two_numbers(3,6)
6 print(z)
7
```

Functions: Tracing 1





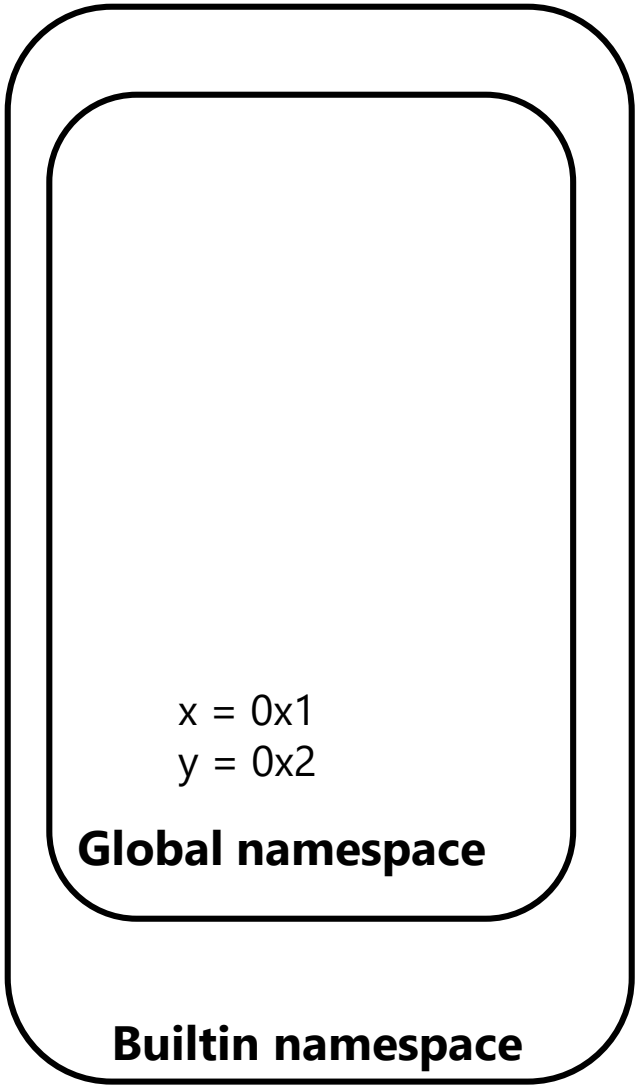

```
x = 2
y = 4
def sum_two_numbers(x,y):
    return x + y
z = sum_two_numbers(3,6)
print(z)
```

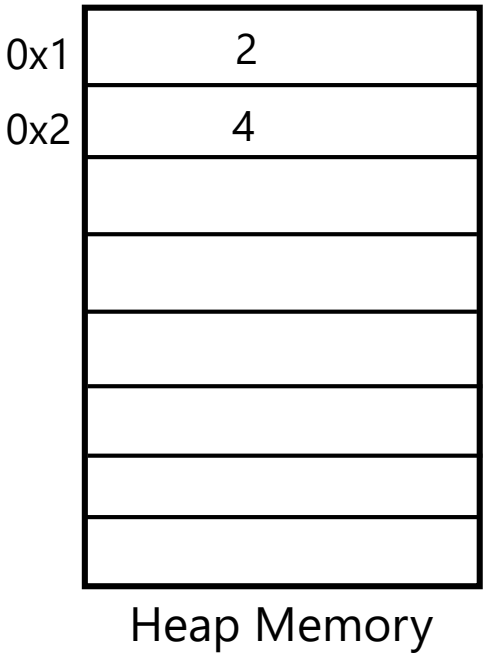
Functions: Tracing 2

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

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

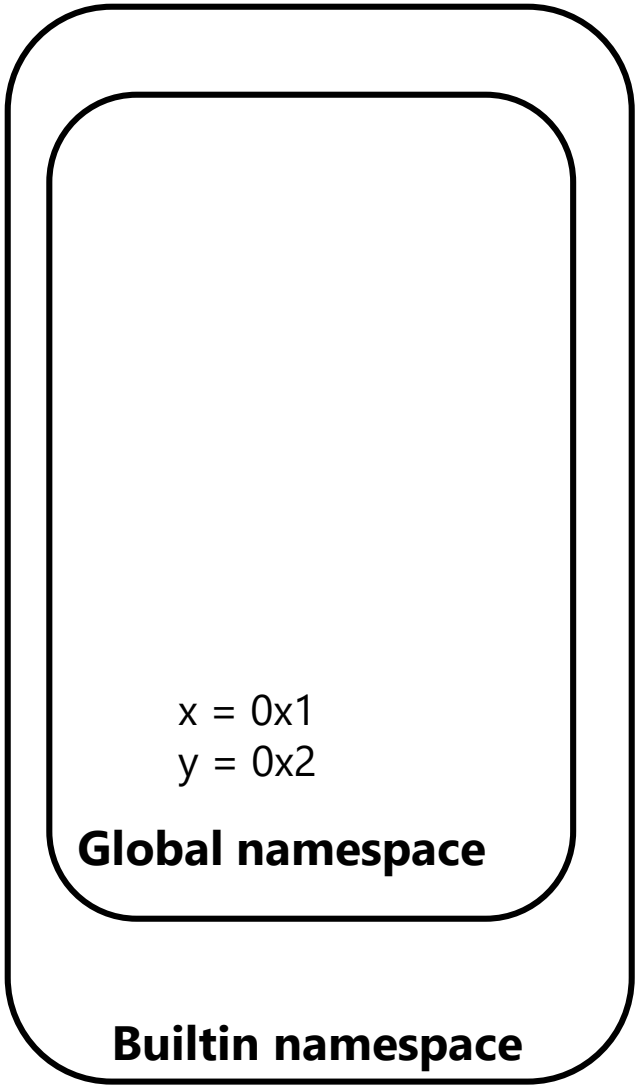
Functions: Tracing 2

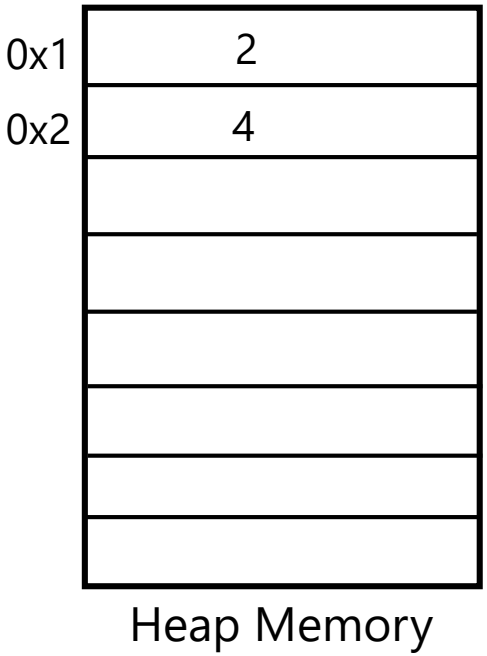





```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Functions: Tracing 2

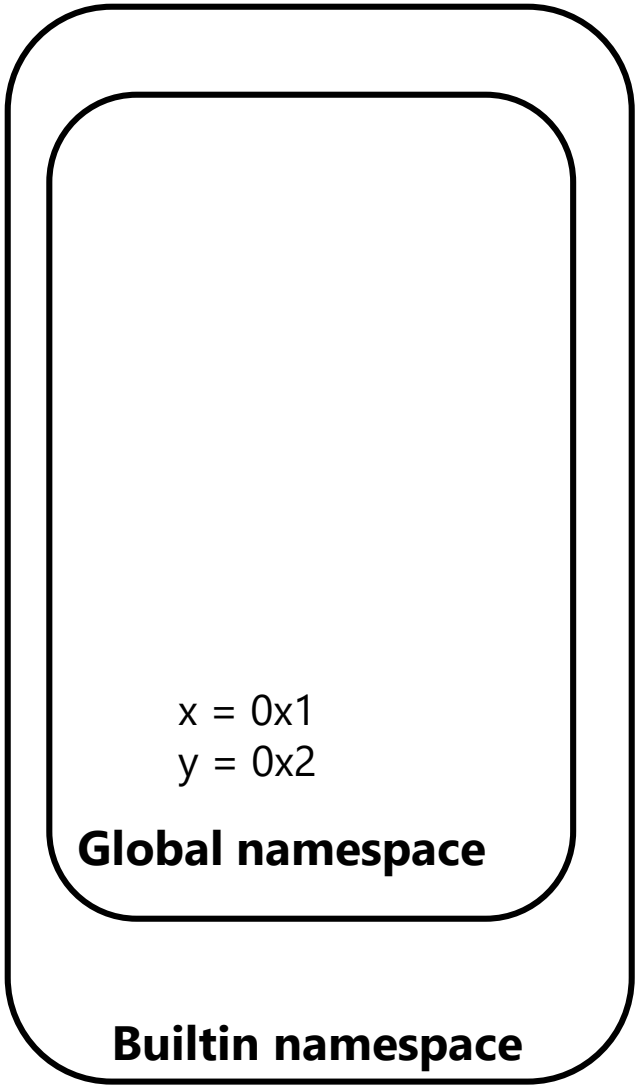


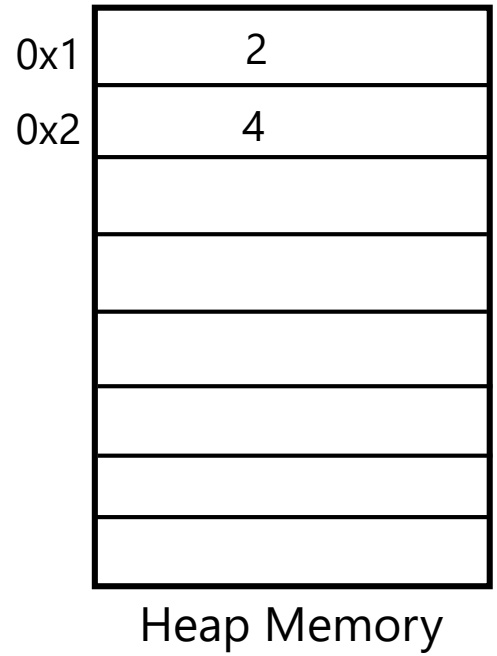



```
x = 2
y = 4
def sum_two_numbers(x):
    return x+y

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

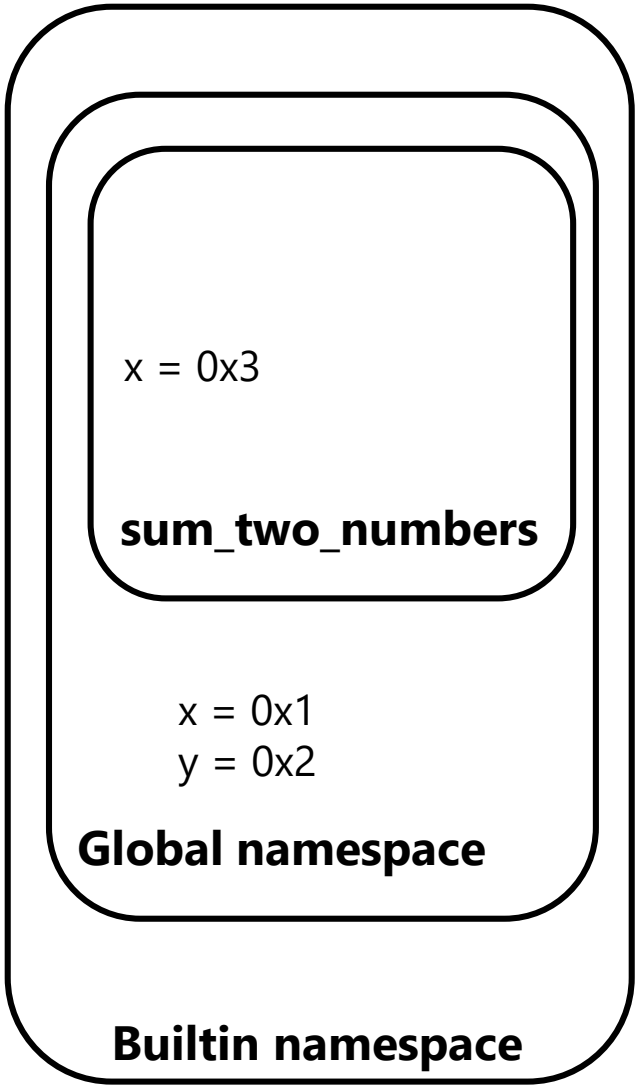
Functions: Tracing 2

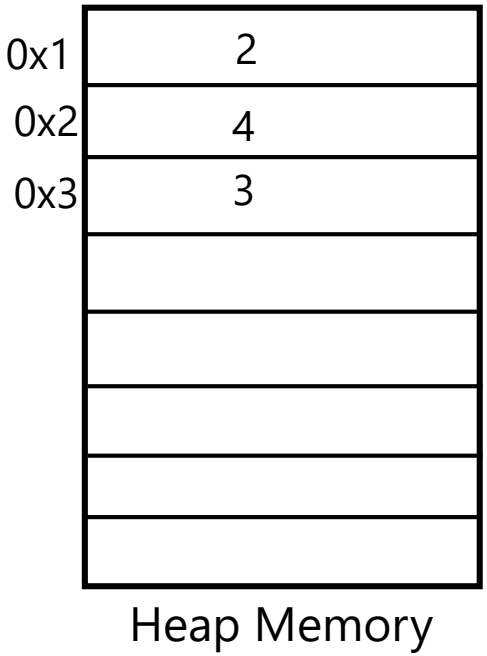





```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

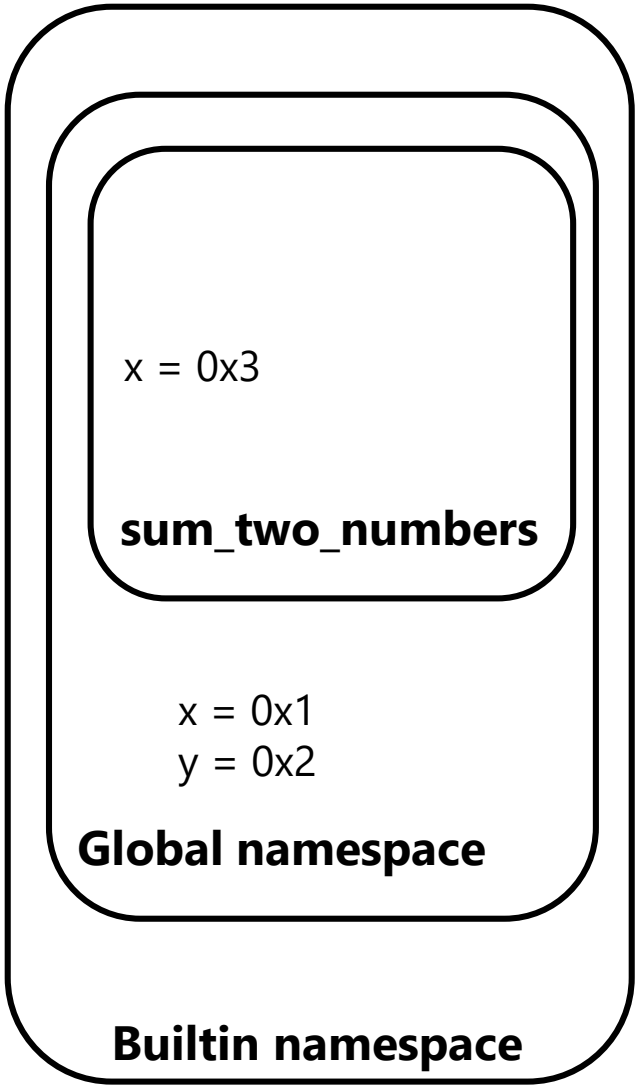
Functions: Tracing 2

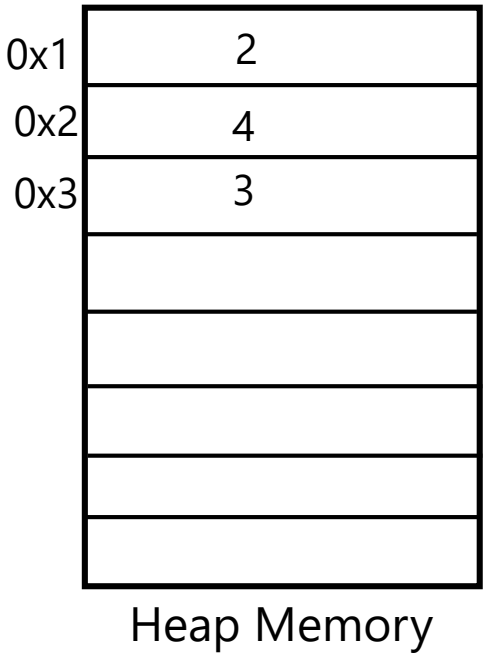





```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

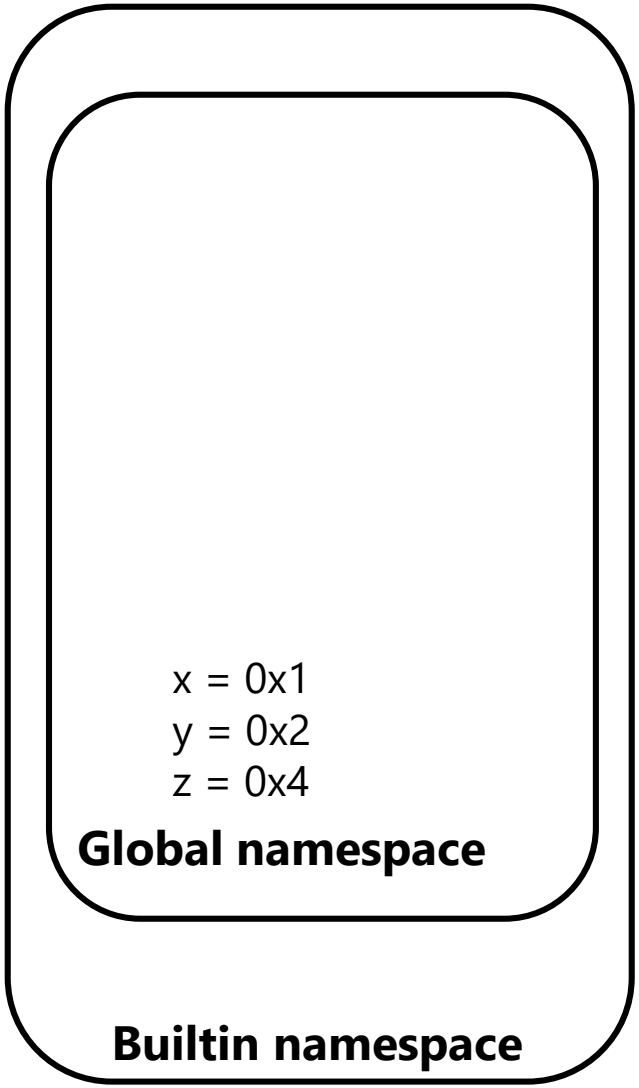
Functions: Tracing 2

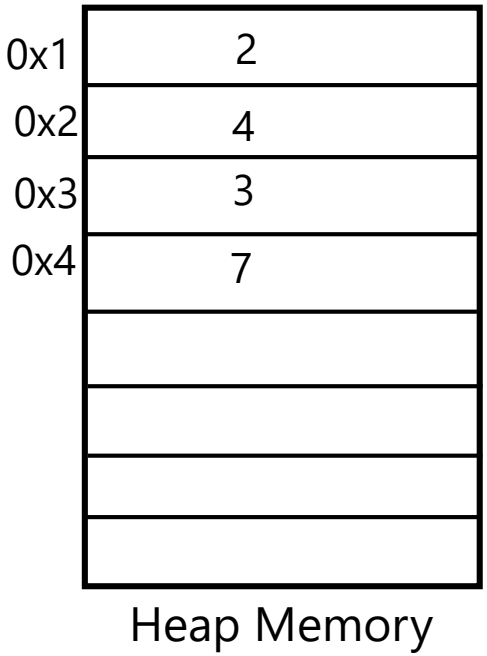





```
1 x = 2
2 y = 4
3 def sum_two_numbers(x):
4     return x+y
5
6 z = sum_two_numbers(3)
7 print(z)
```

Functions: Tracing 2



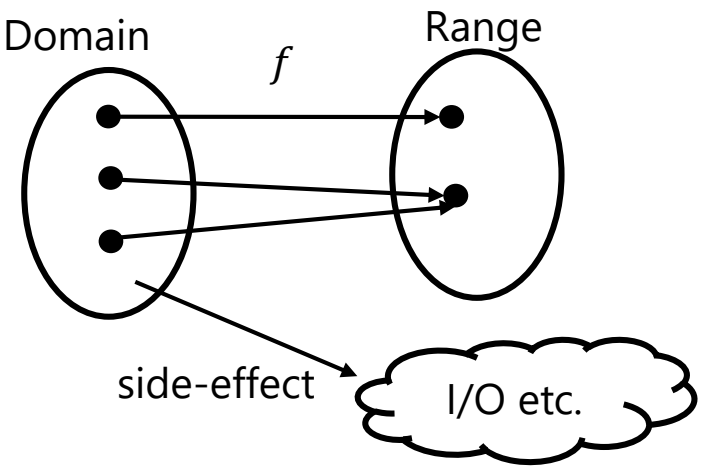



```
x = 2
y = 4
def sum_two_numbers(x):
    return x+y

z = sum_two_numbers(3)
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.



- 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