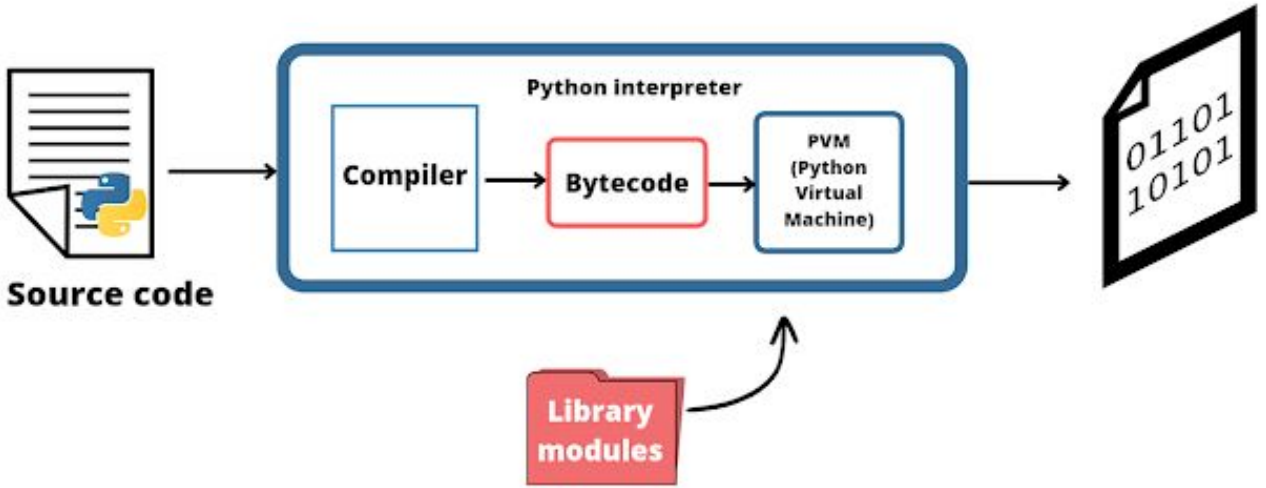
Data and Variables String and Functions

Introduction to Python and Data Science

Instructor: Dr. Sneha Singh, SCEE

Term : August 2025

Data and Algorithms



Command Line Sessions

```
1  Python 3.2.2 (v3.2.2:137e45f15c0b, Sep  3 2011, 17:28:59)
2  [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
3  Type "help", "copyright", "credits" or "license" for more information.
4  >>> # This is a comment.  The interpreter ignores everything
5  ... # after the "#" character.  In the command-line environment
6  ... # the prompt will change to "..." if "#" is the first
7  ... # character of the previous line.
8  ... print("Hello", "World!") # Comment following a statement.
9  Hello World!
10 >>>
```

- The character # as a comment, i.e., it simply ignores this text as it is intended for humans and not the computer.
- Three dots (...) in the command-line environment appears when Python is expecting more input.
- In the IDLE convention, the prompt following a comment is still **>>>,** showing an interactive session

Command Line Sessions: File Editor

```
>>> "Hello World!"
'Hello World!'
>>> print("Have we said enough hellos?")
>>>
Have we said enough hellos?
>>>
```

```
>>> # Can I write Print()?
>>> Print("Hello World!")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'Print' is not defined
>>> # Can I get rid of the parentheses?
>>> print "Hello World!"
  File "<stdin>", line 2
    print "Hello World!"
                ^
SyntaxError: invalid syntax
>>> # Do I need the quotation marks?
>>> print(Hello World!)
  File "<stdin>", line 2
    print(Hello World!)
                  ^
SyntaxError: invalid syntax
```

Bugs

Data and Types: Literals

A literal is code that exactly represents a value, can be:

- Numeric
    - Integer (int)
    - Float (float)
    - Complex (complex)
    - Boolean (bool)
- Non-Numeric
    - Textual/Char (str/char)
    - List (list)
    - Tuples (tup)
    - Dictionary (dic)

Data and Types: Literals

Name	***Type***	***Description***
Integers	int	***Whole numbers***; 3, 5, 69, -2
Floating point	float	***Numbers with decimal point***; 2.3, 5.0, 100.01
Character	str	***A single character string*** `my_char = 'a'  # my_string = "hello"`
Strings	str	***Ordered sequences of characters***; “hello”, “Good Morning”
List	list	***Ordered sequence of objects;*** [100, “hello”, 5.0]
Dictionaries	dict	***Unordered key:value pairs;*** {“Country”**:**”India”, “Name”**:**”Sneha”}
Tuples	tup	***Ordered immutable sequence of objects;*** (100, “hello”, 5.0)
Sets	set	***Unordered collection of unique objects;*** {“a”, “b”}
Booleans	bool	***Logical values indicating;*** TRUE, FALSE

Data and Algorithms

```
>>> type(42)
<class 'int'>
>>> type(-27.345e14)
<class 'float'>
>>> type("Hello World!")
<class 'str'>
>>> type("A") # Single character, double quotes.
<class 'str'>
>>> # Single character, single quotes; looks like int but isn't!
>>> type('2')
<class 'str'>
>>> type("")  # Empty string.
<class 'str'>
>>> type('-27.345e14')  # Looks like float but isn't!
<class 'str'>
```

Python Vocab

Program: *file of code that may contain one or more functions*

Variable: *names that you can assign values to and reuse later on*

Operators: *Mathematical symbols, like +, -, *, /, and ***

Comments: *notes ignored by the computer (#)*

Indentation: *the whitespace (spaces) at the beginning of a code line*

Keywords: *“and”, “print”, and “if”*

Function: *built-in, and “user-defined”. Reusable piece of program*

Python Vocab: Example

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Tue Jun 10 14:10:05 2025
4
5   @author: SnehaPC
6   """
7
8   def reverse_a_number(num):
9
10      reversed_a_number = int(str(num)[::-1])
11      return reversed_a_number
12
13  num = int(input("Enter a number: "))
14  reversed_a_num = reverse_a_number(num)
15  print(f"The reverse of {num} is {reversed_a_num}.")
```

Command Console

```
Enter a number: 456
The reverse of 456 is 654.
```

Variable Explorer

Name	Type	Size	Value
num	int	1	456
reversed_a_num	int	1	654

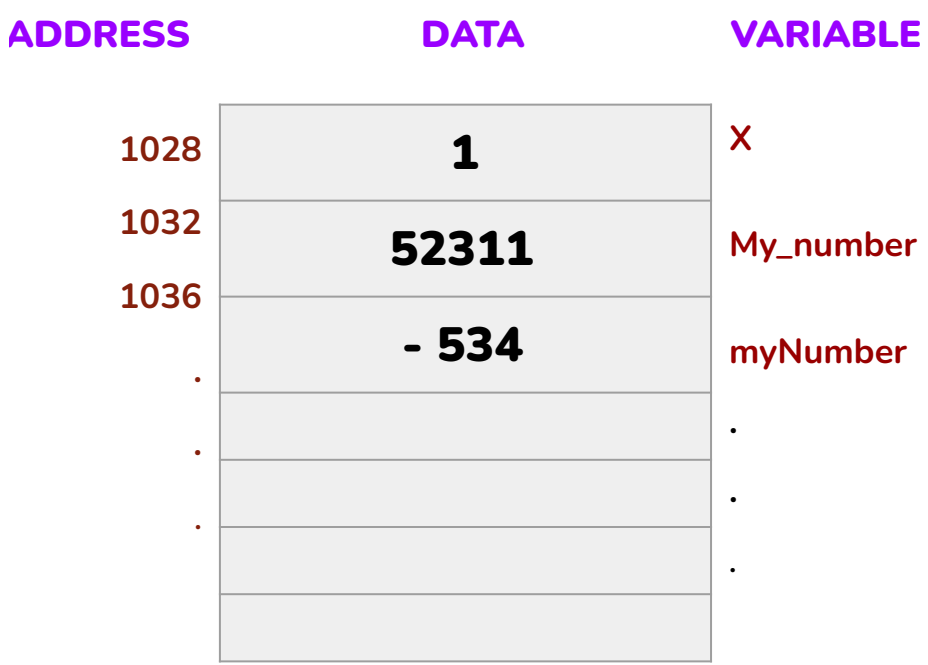
Variables: Memory Allocation

Memory location is given

- Name of a variable
- For programmer's ease

Variable type

- Defines kind of data
- 4 bytes (32 bits) for an integer x≠0, s.t. |x| < 1073741824

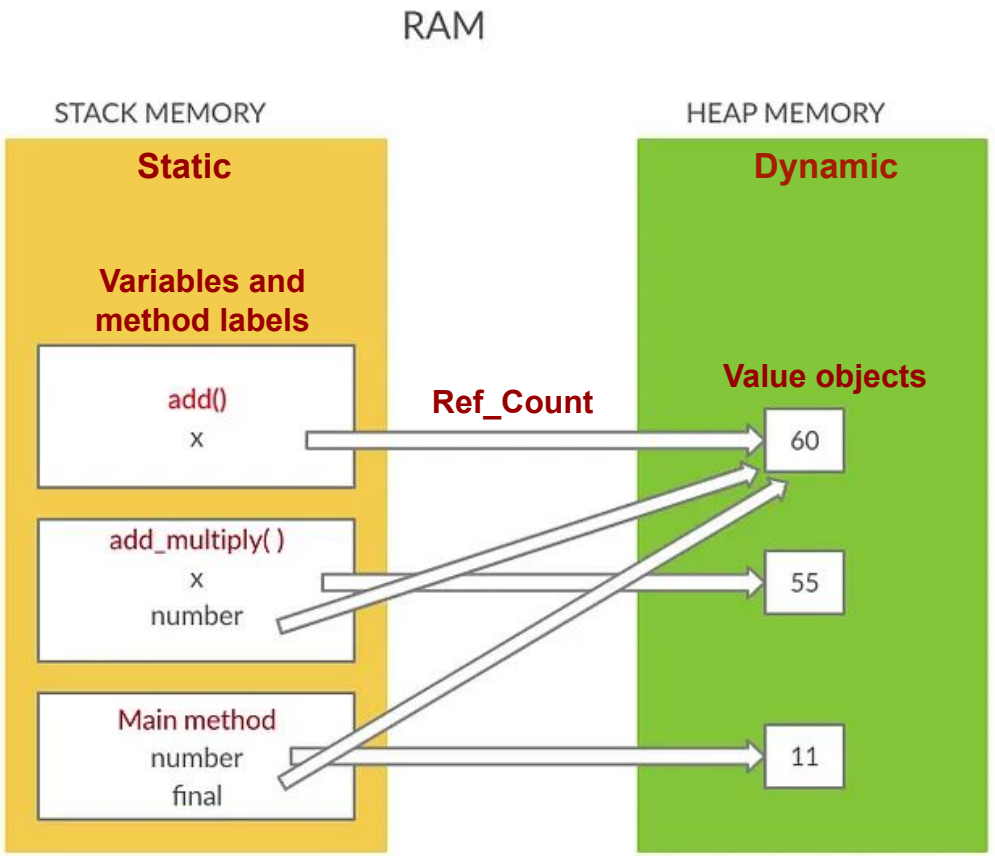


Variables: Memory Allocation

```
>>> def add ( x) :
...     x = x + 5
...     return x
...
>>> def add__multiply ( x) :
...     x = x * 5
...     number = add ( x)
...     return  ( number)
...
```

```
>>> n = 11
>>> out = add__multiply ( n)
>>> out
    60
```

Methods and References are created in the stack memory. Whereas Objects are created in heap memory.



Variable Assignment: Example

```
>>> a = 5
>>> a
5
>>> b = 2*a
>>> b
10
>>> a = 'Hello'
>>> a
'Hello'
>>> b
10
```

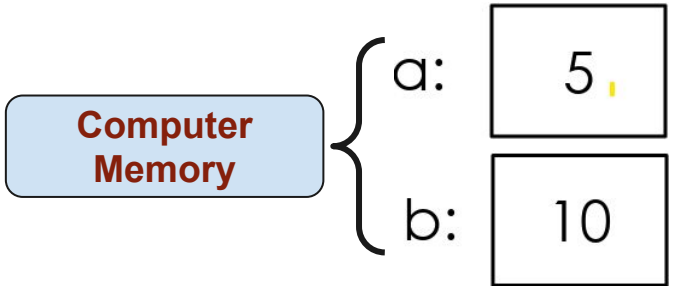
Assignment

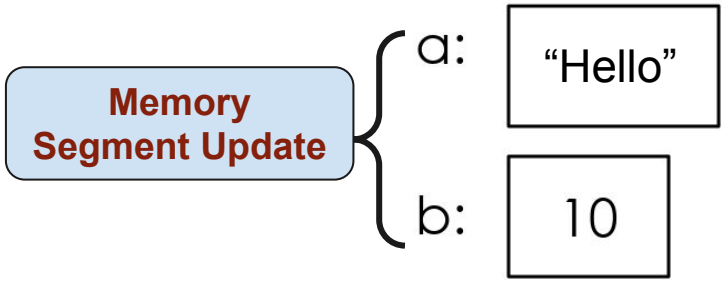
Expression

Response

Overwriting

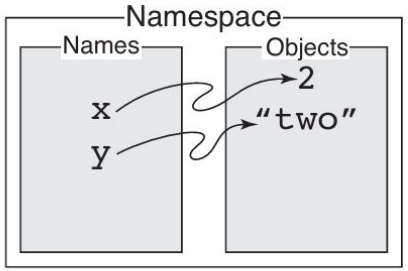
Variable b does not "remember" that its value came from variable a.





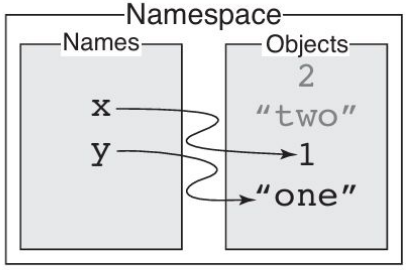
Variable Assignment: Example

```
x, y = 2, "two"
```



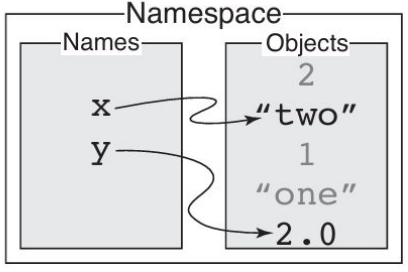
(a)

```
x, y = 2, "two"
x, y = x - 1, "one"
```



(b)

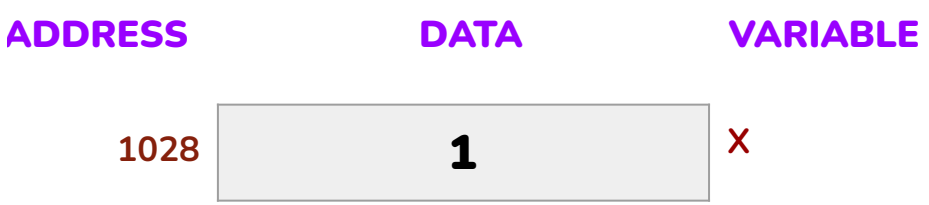
```
x, y = 2, "two"
x, y = x - 1, "one"
y, x = x + 1.0, "two"
```



Variables: Memory Allocation

Memory location is given

- Name of a variable
- For programmer's ease



Variable type

- Defines kind of data
- 4 bytes (32 bits) for an integer x≠0, s.t. |x| < 1073741824
- 8 bytes (64 bits) for float

Command Line

```
>>> x = 1
>>> type(x)
<class 'int'>
```

Variables: Memory Allocation

Memory location is given

- Name of a variable
- For programmer's ease

Variable type

- Defines kind of data
- 4 bytes (32 bits) for an integer x≠0, s.t. |x| < 1073741824
- 8 bytes (64 bits) for float

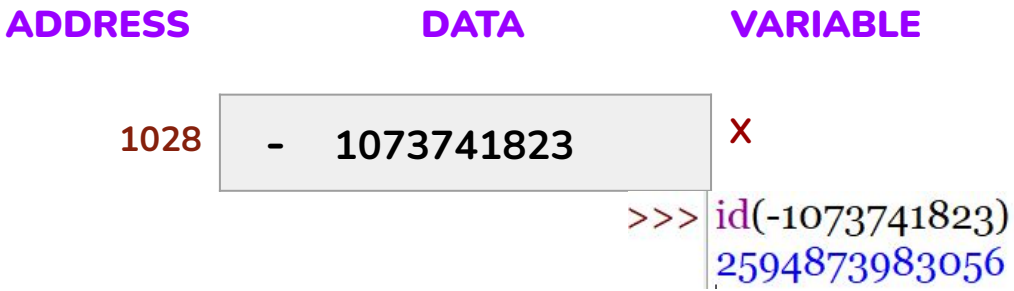
ADDRESS	DATA	VARIABLE
1028	- 1073741823	X

```
>>> import sys
>>> num = 32
>>> sys.getsizeof(num)
28
>>> # Output: Memory size of integer 32
>>> id(num)
140728484644744
>>> # Output: Memory address of the integer object
>>> sys.getsizeof(1073741823)
28
>>> sys.getsizeof(1073741824)
32
>>> bin(1073741824)
'0b1000000000000000000000000000000'
>>> sys.getsizeof(-1073741823)
28
>>> bin(-1073741824)
'-0b1000000000000000000000000000000'
>>>
```

Variables: Memory Allocation

Memory location is given

- Name of a variable
- For programmer's ease



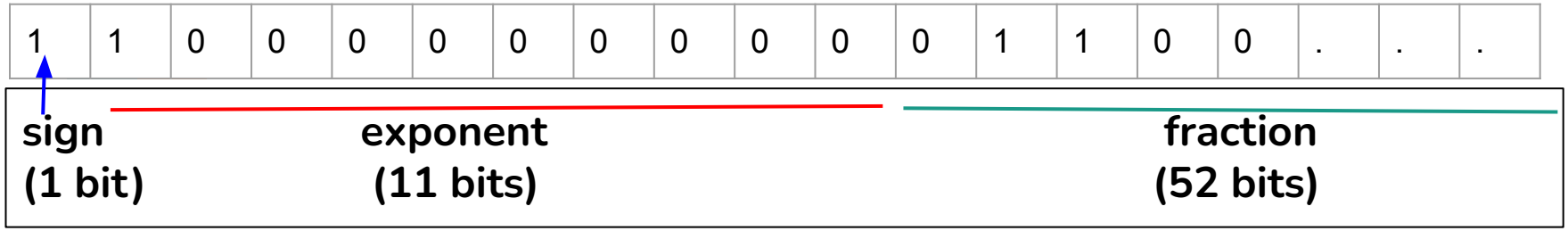
Variable type

- Defines kind of data
- 4 bytes (32 bits) for an integer x≠0, s.t. |x| < 1073741824
- 8 bytes (64 bits) for float

```
>>> x = 3.143
>>> import sys
>>> sys.getsizeof(x)
24
>>> sys.getsizeof(0.0)
24
>>> sys.getsizeof(122152456126214561212532125352.1235213)
24
>>> x = 122152456126214561212532125352.1235213
>>> x
1.2215245612621456e+29
```

Variables: Bit Allocation

IEEE 754 standard



Memory location is given

- Name of a variable
- For programmer's ease

Variable type

- Defines kind of data
- 4 bytes (32 bits) for an integer $x \neq 0$, s.t. $|x| < 1073741824$
- 8 bytes (64 bits) for float

$$n = (-1)^{s} * 2^{e - 1023} * (1+f)$$

$$s = 1$$

$$e = 1*2^{10} + 0*2^{9} + \ldots$$

$$f = 1*2^{-1} + 1*2^{-2} + 0*2^{-3} + \ldots$$

$$n = (-1)^{1} * 2^{1024-1023} * (1+ 0.5 + 0.25)$$
$$= - 3.5$$

Variables: Assignments

Rules for variable names in python:

- Can only contain letters, numbers, and underscores.
- Can not start with a number

Tips:

- Descriptive: my_number for a number, myString for a string etc.
- Consistent: myNumber or my_number
- Traditions: avoid starting with _, start with lowercase
- keep them short

Identifiers: Assignments

In addition to assigning names to values (or objects), there are other entities, such as functions and classes, for which programmers must provide a name, called identifiers.

`one2three`	Valid.
`one_2_three`	Valid.
`1_2_three`	Invalid. Cannot start with a digit.
`one-2-three`	Invalid. Cannot have hyphens (minus signs).
`n31`	Valid.
`n.31`	Invalid. Cannot have periods.
`trailing__`	Valid.
`__leading`	Valid.
`net worth`	Invalid. Cannot have blank spaces.
`vArIaBlE`	Valid.

Keywords

Reserved words that have special meaning and cannot be used as an identifier.

False	None	True	and	as	assert	break
class	continue	def	del	elif	else	except
finally	for	from	global	if	import	in
is	lambda	nonlocal	not	or	pass	raise
return	try	while	with	yield		

The 33 keywords in Python 3.

import keyword;

print(keyword.kwlist).

Keywords

So, can we have a variable named print ?

```
>>> print = 10        # Assign 10 to print.
>>> print             # Check what print is.  Assignment worked!
10
>>> print("Hello!") # Try to print something...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

**If an identifier had previously been used, the old association is forgotten when a new value is assigned to it.*

Keyword Recovery: Recover the built-in function by deleting, using the del() function,

Questions

- How do we define a variable in Python, and why do we use one ?
- What types of values can a variable hold in Python? Give examples for different types.
- How did the stored program concept change the way computers were used and programmed compared to earlier machines?
- Can 0s be variable name?
- Will following statement give error in python, why?
- _3 = 3 ?

Operators in Python

Operations and Operators

Operators can be defined as symbols that are used to perform operations on operands; can be ***unary or binary*** operators.

These operators are classified into four categories:

- ***Arithmetic*** (+, -, *, / , // ,%, **)
- ***Relational*** (== , !, =, < , >, <=, >=, is, in)
- ***Logical*** (and, not, or)
- ***Bitwise*** (&, |, ~, ^, <<, >>)

Operators uses a header-like notation called a “prototype”, includes the symbol/name of the operator, the type(s) of its operand(s), and the type of result it returns; ***< (int, int) -> bool***

Arithmetic Operations

Operators	Description	Example
+	perform addition of two number	a+b
-	perform subtraction of two number	a-b
/	perform division of two number	a/b
*	perform multiplication of two number	a*b
%	Modulus = returns remainder	a%b
//	Floor Division = remove digits after the decimal point	a//b
**	Exponent = perform raise to power	a**b

Arithmetic Operations: Examples

```
>>> 4 - 5          # Difference of two ints.
-1
>>> 5 + 4          # Sum of two ints.
9
>>> 5 + 4.         # Sum of int and float.
9.0
>>> 5 + 4 * 3      # * has higher precedence than +.
17
>>> (5 + 4) * 3    # Parentheses can change order of operations.
27
>>> 5 / 4          # float division of two ints.
1.25
>>> 10 / 2         # float division of two ints.
5.0
>>> 20 / 4 / 2     # Expressions evaluated left to right.
2.5
>>> (20 / 4) / 2 # Parentheses do not change order of operation here.
2.5
>>> 20 / (4 / 2) # Parentheses do change order of operation here.
10.0
```

Additional Arithmetic Operations: Exponents

```
>>> 2 ** 3
8
>>> 2 ** 3 ** 4     # Operator is right associative.
2417851639229258349412352
>>> 3 ** 4
81
>>> (2 ** 3) ** 4   # Use parentheses to change order of operation.
4096
>>> 3 ** 400        # A big number.  Exact value as an int.
705507910865533257124642715759347962165079496127873157628712232092620855515829341565792985294471341581549523348253559118669297930718245666941450844545352570279602853237603131924432833340880001
>>> 3.0 ** 400      # A big number.  Approximate value as a float.
7.055079108655333e+190
```

Additional Arithmetic Operations: Floor Division

*The operator / always represents float division while // is always used for **floor division**.*

The quotient is a whole number.

```
>>> time = 257              # Time in seconds.
>>> minutes = time // 60 # Number of complete minutes in time.
>>> print("There are", minutes, "complete minutes in", time, "seconds.")
There are 4 complete minutes in 257 seconds.
>>> 143 // 25
5
>>> 143.4 // 25
5.0
>>> 9 // 2.5
3.0
```

Arithmetic Operations: Precedence/Associativity

Preference

Operators	Associativity
( ), [ ], { }	left to right
func( ), array[ ]	left to right
**	right to left
/, //, %, *	left to right
+, -	left to right
<, <=, >, >=, !=, ==	left to right
not	left to right
and	left to right
or	left to right

Arithmetic Operations: Questions?

1.
```
x += 1
```

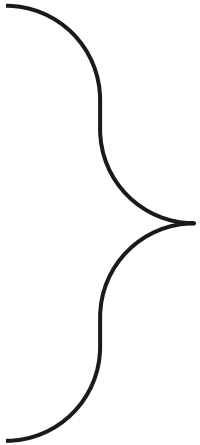
Augmented Assignment (for interactive environment): x = x + 1

2.
```
print(7 + 23)
```

3.
```
x = (3 +
4)

x = 3 + \
4

x = """3 +
4"""
```



Are they Equivalent ?

Formatting Code

Guideline	Example	Common Mistakes
Parentheses: no space before or after.	`print("Go team!")`	`print ("Go team!")` `print( "Go team!" )`
Commas: no space before, one space after.	`print("Hello", name)`	`print("Hello" , name)` `print("Hello",name)`

Assignment: one space before and after the =.	`name = input("Your name? ")`	`name=input("Your name? ")`
		`name= input("Your name? ")`
		`name =input("Your name? ")`

Concatenation: one space before and after the +.	`print("Hi", name + "!")`	`print("Hi", name+"!")`
		`print("Hi", name+ "!")`
		`print("Hi", name +"!")`

Arithmetic: use space to show lower precedence.	`x**2 + 5*x - 8`	`x ** 2 + 5 * x - 8`
		`x ** 2+5 * x-8`
		`x**2+5*x-8`

Relational Operators

Relational operators always return a boolean result that indicates whether some relationship holds between their operands

Operators	Description	Example
==	Equal to, return true if a equals to b	a == b
!=	Not equal, return true if a is not equals to b	a != b
>	Greater than, return true if a is greater than b	a > b
>=	Greater than or equal to , return true if a is greater than b or a is equals to b	a >= b
<	Less than, return true if a is less than b	a < b
<=	Less than or equal to , return true if a is less than b or a is equals to b	a <= b

Relational Operators: Identity and Inclusion

Compares the memory locations of two objects

Operators	Description	Example
is	returns true if two variables point the same object/value, else false	a is b
is not	returns true if two variables point the different object/value, else false	a is not b

Validate the membership of the query object within the given sequence such as string, list or tuples

Operators	Description	Example
in	return true if value exists in the sequence, else false.	a in list
not in	return true if value does not exists in the sequence, else false.	a not in list

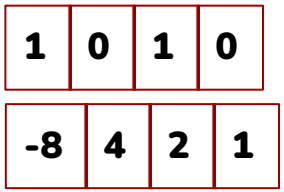
Logical Operators

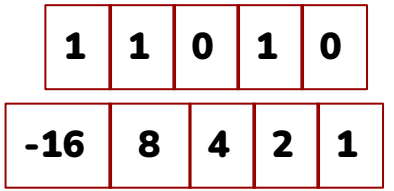
Operators	Description	Example
and	return true if both condition are true	x and y
or	return true if either one or both condition are true	x or y
not	reverse the condition	not(a>b)

a	b	a and b	a or b	not b	a != b (exclusive or)
False	False	False	False	True	False
False	True	False	True	False	True
True	False	False	True	*	True
True	True	True	True	*	False

1’s Complement

- Signed representation
- Swap 0 with 1 and 1 with 0, and then append 1 on left as sign bit
- ~5 -> 101 -> swap -> 010 -> append sign bit -> 1010
- Binary to Integer:





- -8*1 + 4*0 + 2*1 + 1*0 = -6
- Reverse of 1’s complement:
- Remove 1 from left, and then swap 0 with 1 and 1 with 0,
- ~~5 = 5

2's Complement

- ➔ Signed representation
- ➔ Swap 0 with 1 and 1 with 0, and then append 1 on left as sign, then add 1
- ➔ ~5 -> 101 -> swap -> 010 -> append 1 -> 1010 -> then add 1 -> 1011
- ➔ Binary to Integer:
- ➔ Reverse of 2's complement:
- ➔ Subtract 1, Remove 1 from left, and then swap 0 with 1 and 1 with 0,
- ➔ ~[(~5 + 1) - 1] = 5

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	False

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & True	False
OR	|	True | False	True

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	False
OR	|	True | False	True
XOR	^	True ^ False	True

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	False
OR	|	True | False	True
XOR	^	True ^ False	True
NOT	~	~ True	False

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	False
OR	|	True | False	True
XOR	^	True ^ False	True
NOT	~	~ True	False
Left Shift	<<	True << 1	
		True << 3	

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	False
OR	|	True | False	True
XOR	^	True ^ False	True
NOT	~	~ True	False
Left Shift	<<	True << 1	2
		True << 3	8

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	True & False	False
OR	|	True | False	True
XOR	^	True ^ False	True
NOT	~	~ True	False
Left Shift	<<	True << 1	2
		True << 3	8
Right Shift	>>	True >> 1	0

Bitwise Operations

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7
XOR	^	2 ^ 7	5

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7
XOR	^	2 ^ 7	5
NOT	~	~ 5	?

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7
XOR	^	2 ^ 7	5
NOT	~	~ 5	-6

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7
XOR	^	2 ^ 7	5
NOT	~	~ 5 00000101 11111010	-6

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7
XOR	^	2 ^ 7	5
NOT	~	~ 5	-6
Left Shift	<<	2<< 1	4

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

Bitwise Operations

Operation	Python Operator	Example	Evaluates To
AND	&	2 & 3	2
OR	|	2 | 5	7
XOR	^	2 ^ 7	5
NOT	~	~ 5	-6
Left Shift	<<	2<< 1	4
Right Shift	>>	5 >> 1	2
		5 >> 2	1

```
>>> bin(2)
'0b10'
>>> bin(5)
'0b101'
>>> bin(7)
'0b111'
>>> bin(3)
'0b11'
>>> bin(-6)
'-0b110'
```

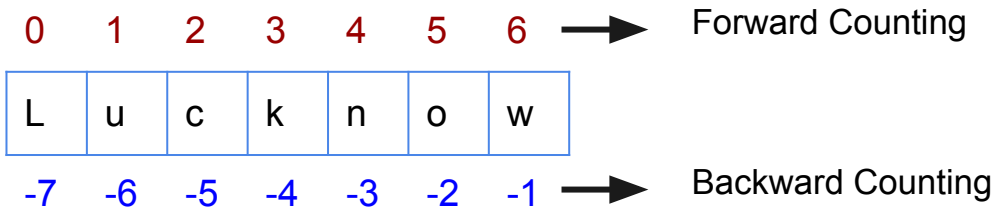
Questions

- What will the bitwise operation XOR return when applied to ‘12’ and ‘25’,
- What will the bitwise OR and AND operations return when applied to ‘12’ and ‘25’ ?
- What would be 1’s Complement range in 4-bits ?
- What is the result of right-shifting & left-shifting the number 'x' by 2 positions ? Answer in terms of x, 2^n.
- What is the relationship between the bitwise NOT operation and the two's complement of a number in Python ?

String Operations

- String: Sequence or List of characters
- Starting from index value 0 to N-1, for a string of length N.
- Enclosed inside the quotes; single, double, triple.

Data = “Lucknow”



a = "Hello, World!"

print(a[1]) ?

Escape sequence	Meaning	Example	Screen output
\n	A newline character that indicates the end of a line of text.	`print("Escape\nsequence!")`	`Escape` `sequence!`
\t	A tab character; useful for indenting paragraphs or aligning text on multiple lines.	`print("Escape\tsequence!")`	`Escape  sequence!`
\'	A single quote; an alternative to enclosing the string in double quotes.	`print('I\'ll try my best!')`	`I'll try my best`
\"	A double quote; an alternative to enclosing the string in single quotes.	`print("I heard you said \"Yes\"")`	`I heard you said "Yes"`
\\	A backslash character.	`print("This prints a \\")`	`This prints a \`

Special Characters

String Operations: Formatted String

- ***Concatenation*** consists of combining two strings end to end with no intervening characters
- Concatenation is built into Python in the form of the **+** operator.

"la" + "te" = "late"

- Python redefines the ***** operator for strings to indicate ***Repetition***.
- The expression *s * n* indicates *n copies of the string s concatenated together.*

*"la" * 3 = "la" + "la" + "la"*

String Operations: Example

```
>>> print("+ adds two numbers in python" + " bu
t also concatenates two" + " or more strings.")
+ adds two numbers in python but also concatena
tes two or more strings.
```

```
>>> print("I want to emphasize agian and again:
" + "use interactive mode as calculator... "*3)
I want to emphasize agian and again: use intera
ctive mode as calculator... use interactive mod
e as calculator... use interactive mode as calc
ulator...
```

String Operations: F String

- A *formatted string literal (or f-string)* is a string literal that is prefixed with ***"f"*** or ***"F"***. A replacement field is an expression in curly braces ***({ })*** inside an f-string.

  What is the output of the following code?

  animal = "dog"

  says = "bark"

  print(*f"My {animal} says {says} {says} {says}"*)

  Output: My dog bark bark bark

Format	Description	Example	Result
`d`	Decimal integer (default integer format).	`f"{12345678:d}"`	`'12345678'`
`,d`	Decimal integer, with comma separators.	`f"{12345678:,d}"`	`'12,345,678'`
`10d`	Decimal integer, at least 10 characters wide.	`f"{12345678:10d}"`	`'  12345678'`
`010d`	Decimal integer, at least 10 characters wide, with leading zeros.	`f"{12345678:010d}"`	`'0012345678'`
`f`	Fixed-point (default is 6 decimal places).	`f"{math.pi:f}"`	`'3.141593'`
`.4f`	Fixed-point, rounded to 4 decimal places.	`f"{math.pi:.4f}"`	`'3.1416'`
`8.4f`	Fixed-point, rounded to 4 decimal places, at least 8 characters wide.	`f"{math.pi:8.4f}"`	`'  3.1416'`
`08.4f`	Fixed-point, rounded to 4 decimal places, at least 8 characters wide, with leading zeros.	`f"{math.pi:08.4f}"`	`'003.1416'`

Functions

Functions

```
>>> print("hello world")
hello world

>>> N = input("give me a number")
give me a number
>>> print(N)
10
```

Functions

```
>>> print("hello world")
hello world
```

- The **print ( )** function displays output to the user.
- **Output** is the information or result produced by a program.

Syntax : *print(value(s), sep= ' ', end = '\n', file=file, flush=flush)*

Functions

```
>>> N = input("give me a number")
give me a number
>>> print(N)
10
```

Whatever you enter as input, the input function converts it into a string.

Prompt

- The **input ( )** function is what a user enters into a program.
- A **prompt** is a short message that indicates the program is waiting for input.

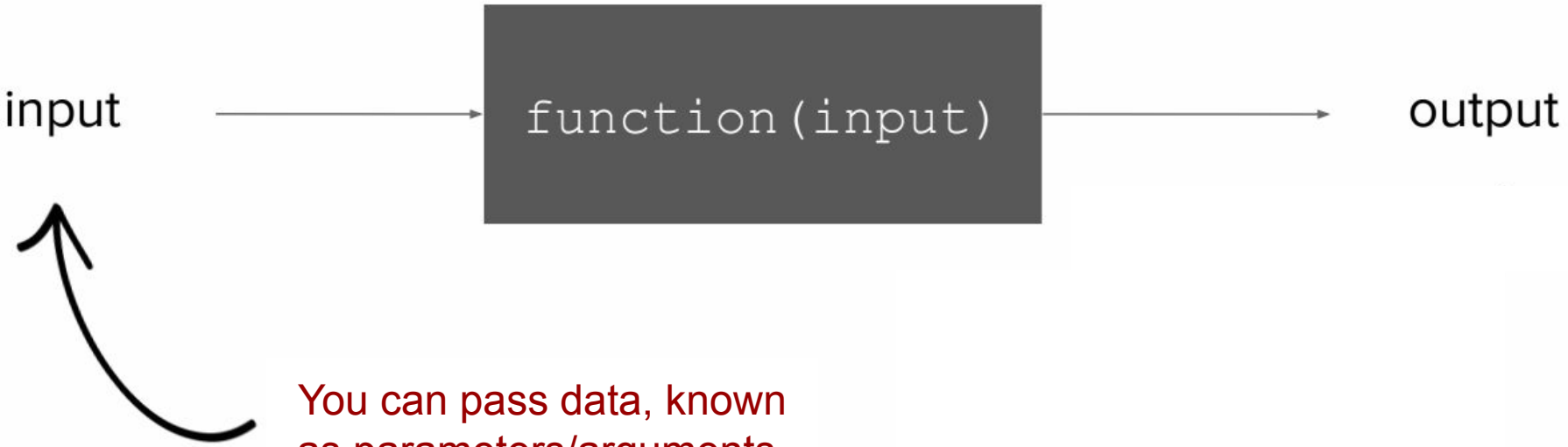
Syntax : *input(prompt)*

Function: Analogy

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

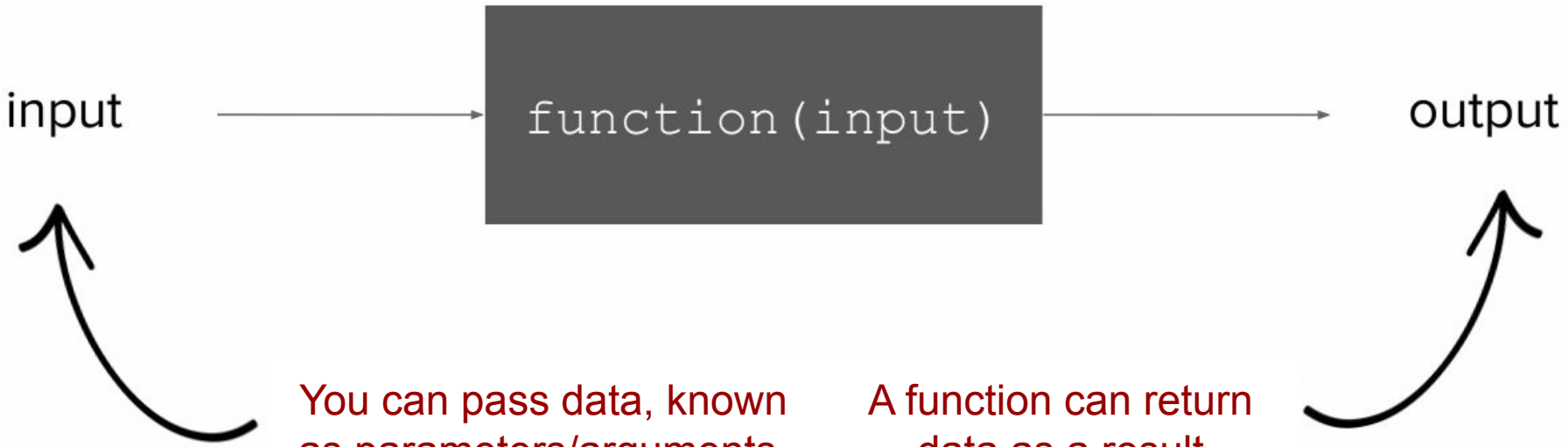
Function: Analogy

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



Function: Analogy

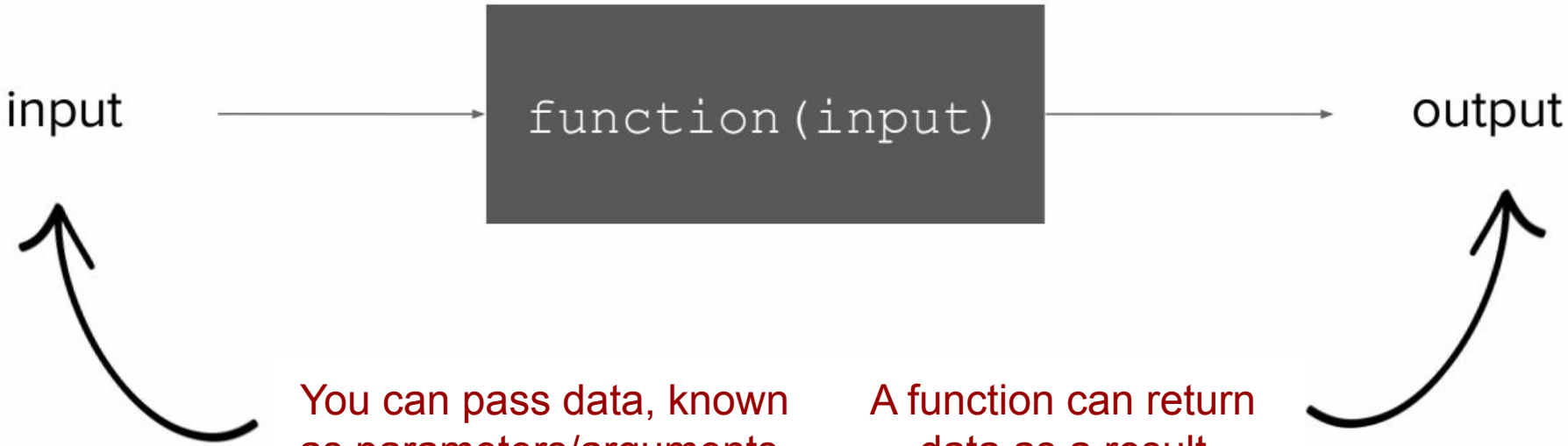
A function is a block of code which only runs when it is called.



Function: Analogy

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

"Do Something"



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

A function can return data as a result

Function: Analogy

- **Arguments**
  - Parentheses
  - Separated by commas
- **Output**
- **Printing multiple arguments in same line and auto conversions**
- **Return the object**

A function is not executed when it is defined.

Defining A Function: Start

```
def <function_name>(<parameter_list>):
    <body>
```

```
def get_final_answer(filename):
    """Documentation String"""
    line1

    line2

    return total_counter
```

Defining A Function: Start

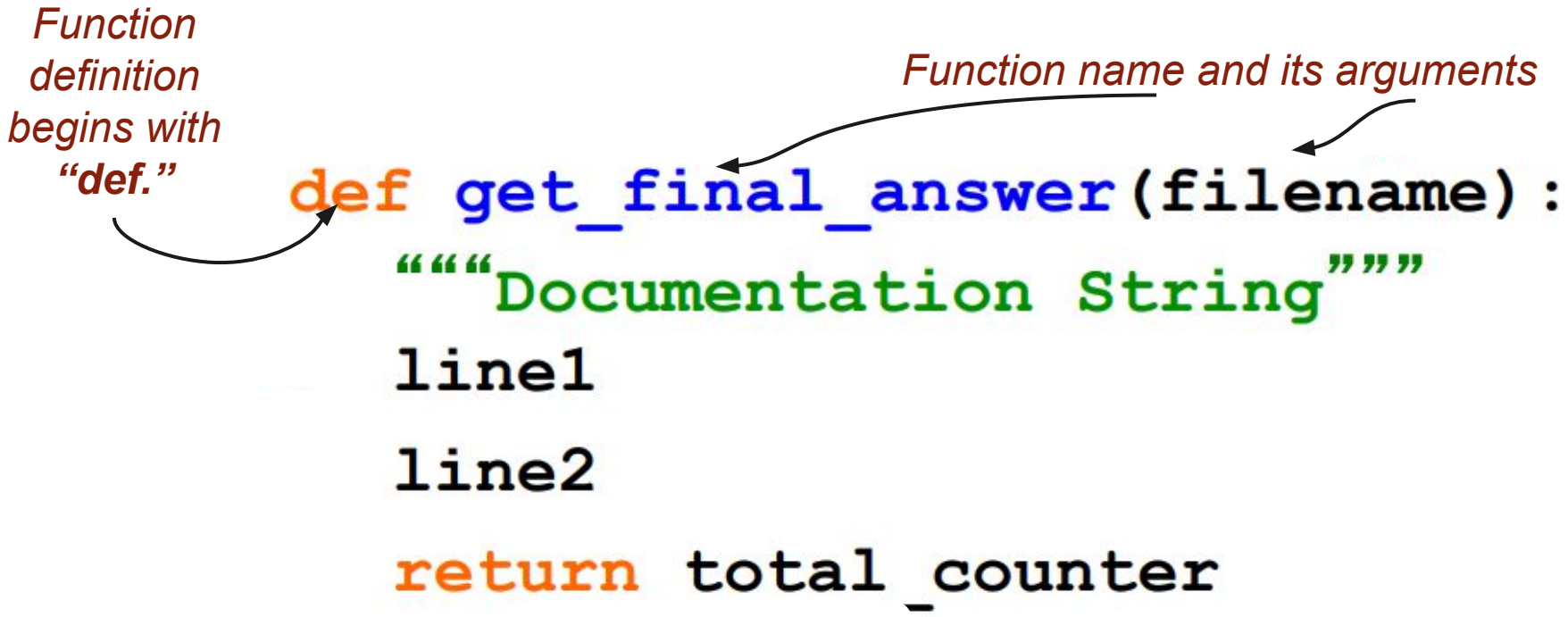
- ***def*** is a key word used to “define” a block of executable code Code within the def block is not available to the interpreter until called.
- ***def*** creates a function object and assigns the object to the name of the function.
- ***def*** can appear as a separate code block in the same python file, or inside a loop or condition or even within another function (enclosed).
- ***lambda*** creates a function object as well, but it has no name. Instead it simply returns the result of the function.

Defining A Function: Start

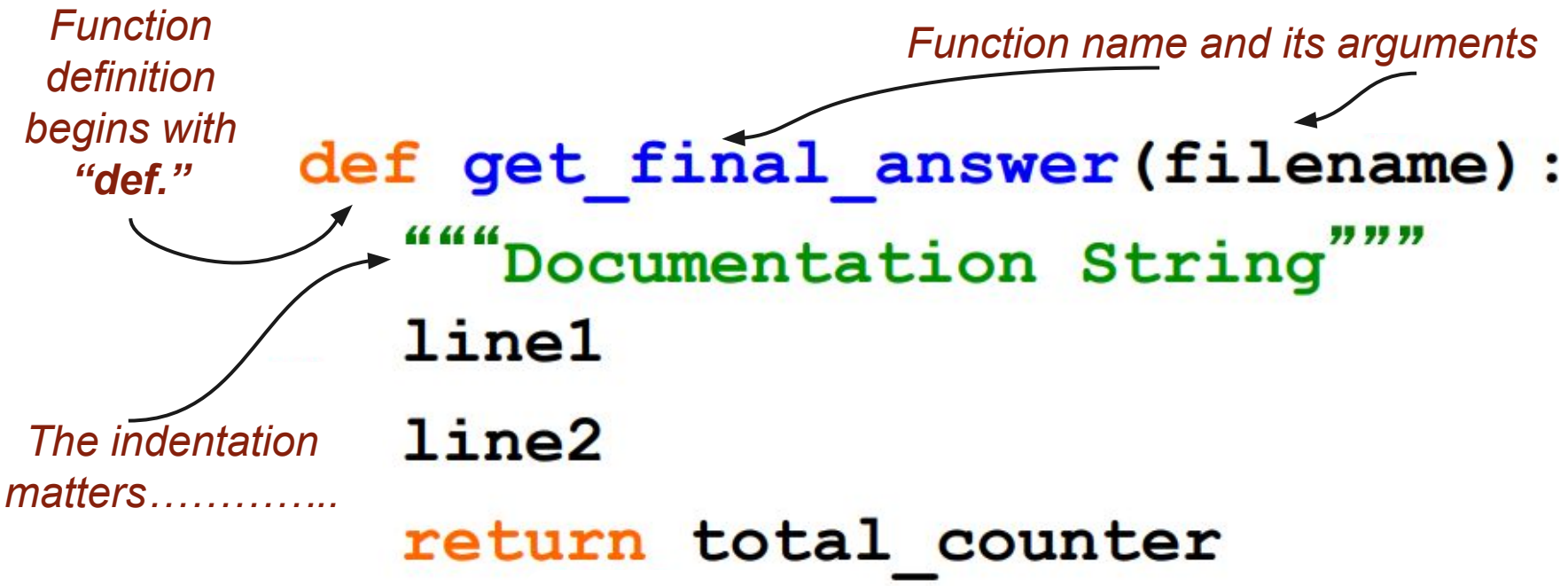
Function definition begins with ***"def."***

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
```

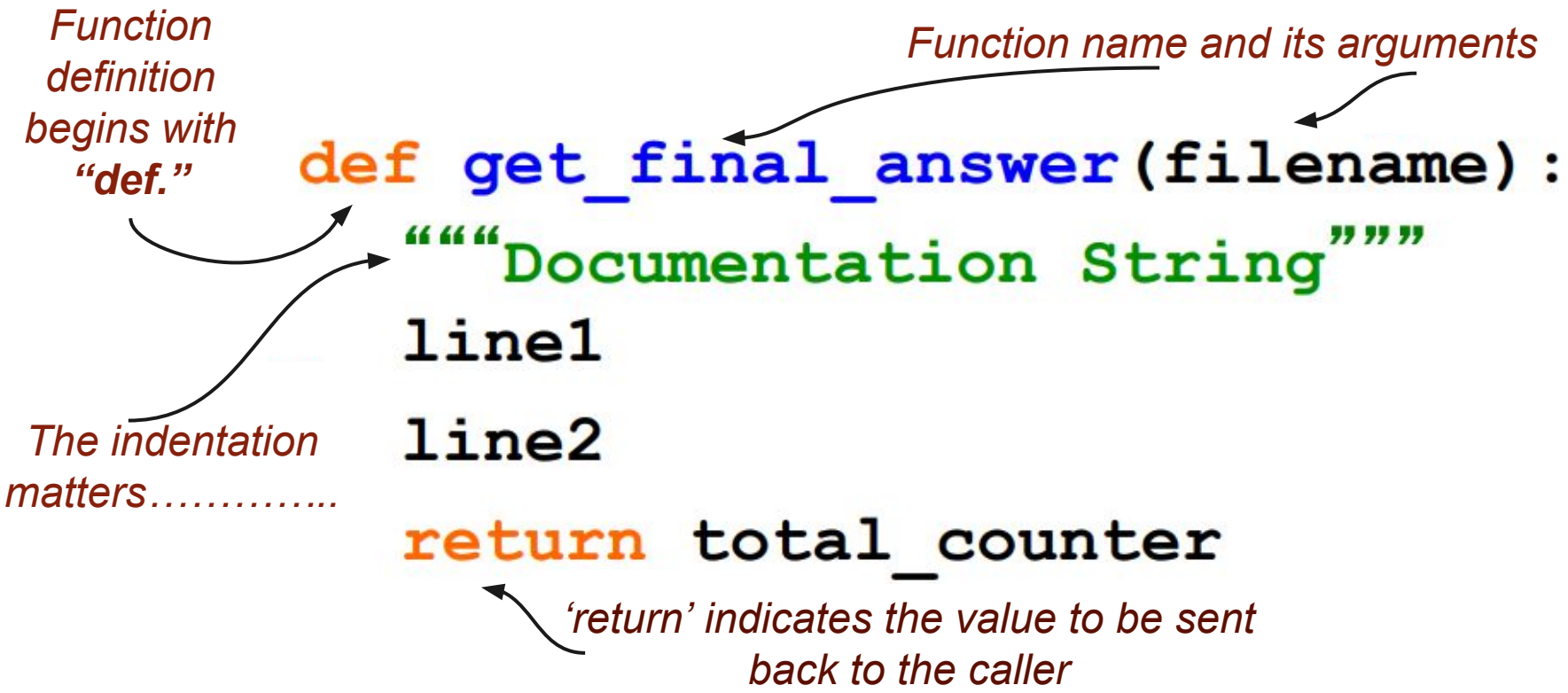
Defining A Function: Start



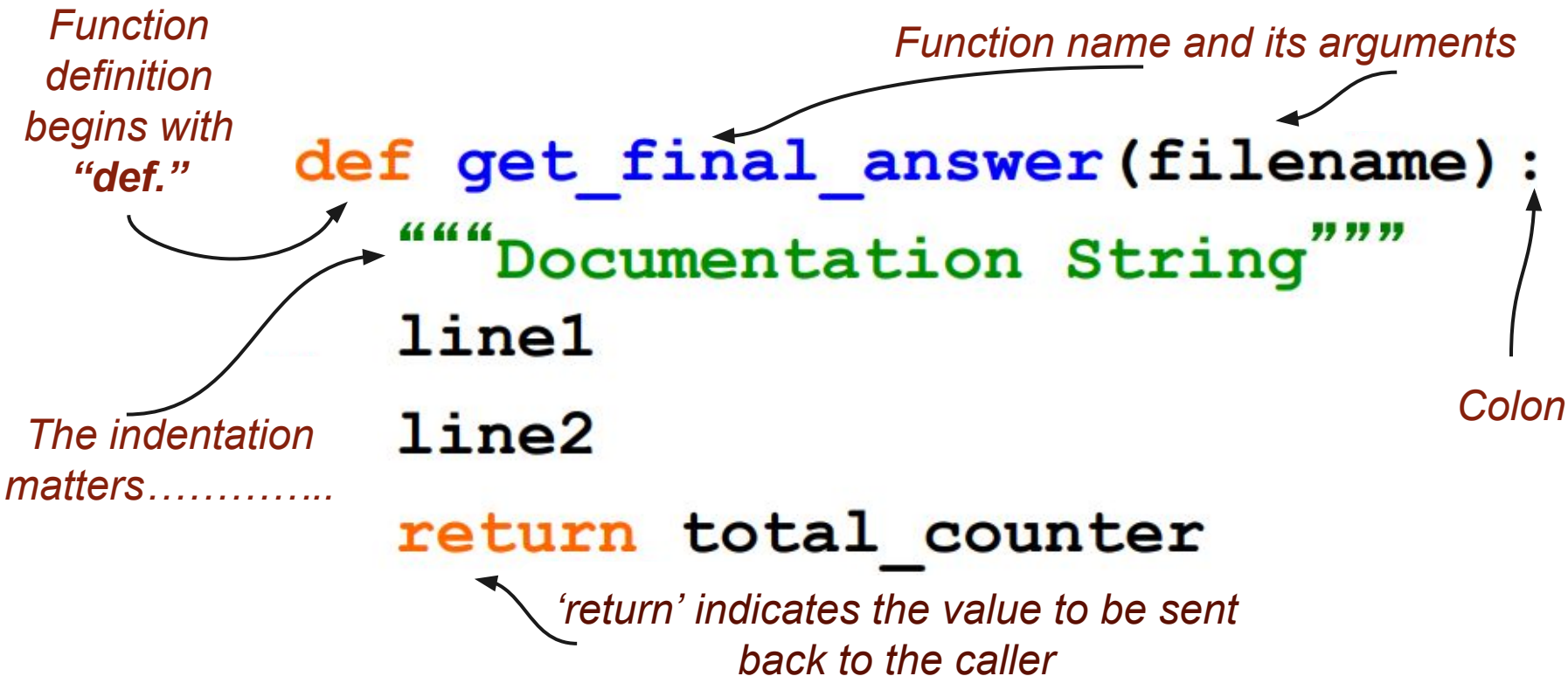
Defining A Function: Start



Defining A Function: Start



Defining A Function: Start



Control flow: Call & Return

- When a function is called the flow of the main program stops and sends the “control” to the function.
- No code in the main program will be executed as long as the function has the control.
- Function “returns” the control to the main program using the return statement.
- Every function has an implicit return statement.
- We can use the return statement to send a value or object back to the main function.
- Since return can send back any object you can send multiple values by packing them into a tuple.

Calling A Function

The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- *Parameters in Python are Call by Assignment*
- *Old values for the variables that are parameter names are hidden, and these variables are simply made to refer to the new values*

Calling A Function

```
#userDefinedNewLine
#prints a new line between two lines
#using user defined function: newLine()
#written by user name on dd/mm/yyyy

def newLine():
    print()

print('first line')
newLine()
print('second line')
```

Output:

```
first line

second line
```

Calling A Function

```
#userDefinedThreeLines
#prints three lines between two lines
#using user defined function: newLine()
#written by user name on dd/mm/yyyy

def newLine():
    print()

def threeLines():
    newLine()
    newLine()
    newLine()

print('first line')
threeLines()
print('second line')
```

Output:

```
first line


second line
```

Calling A Function

```
#userDefinedAnyLines
#prints any number lines between two lines
#using user defined function: newLine()
#written by user name on dd/mm/yyyy

def newLine():
    print('*')

def anyLines(n):
    for i in range(0,n):
        newLine()


print('first line')
anyLines(4)
print('second line')
```

Output:

```
first line
*
*
*
*
second line
```

Function Without Return

All functions in Python have ***a return value, even if no return line inside*** the code

- Functions without a return, return the special value ***None***
- ***None*** is a special constant in the language
- ***None*** is used like *NULL*, void, or nil in other languages
- ***None*** is also logically equivalent to *False*
- The interpreter's *REPL* doesn't print None

If no return statement is present, execution continues until the end of the body, at which point execution returns to the point in the program where the function was called.

Function: Examples

```
>>> def get_wh():
...     """Obtain weight and height from user."""
...     weight = float(input("Enter weight [pounds]: "))
...     height = float(input("Enter height [inches]: "))
...     return weight, height
...
>>> def calc_bmi(weight, height):
...     """
...     Calculate body mass index (BMI) for weight in
...     pounds and height in inches.
...     """
...     return 703 * weight / (height * height)
...
>>> def show_bmi(bmi):
...     print("Your body mass index is:", bmi)
...
>>> w, h = get_wh()
Enter weight [pounds]: 280
Enter height [inches]: 72
>>> bmi = calc_bmi(w, h)
>>> show_bmi(bmi)
Your body mass index is: 37.9706790123
```

Function Call	Output
float("3.14")	3.14
int("41")	41
str(12.312)	'12.312'

Function: Examples

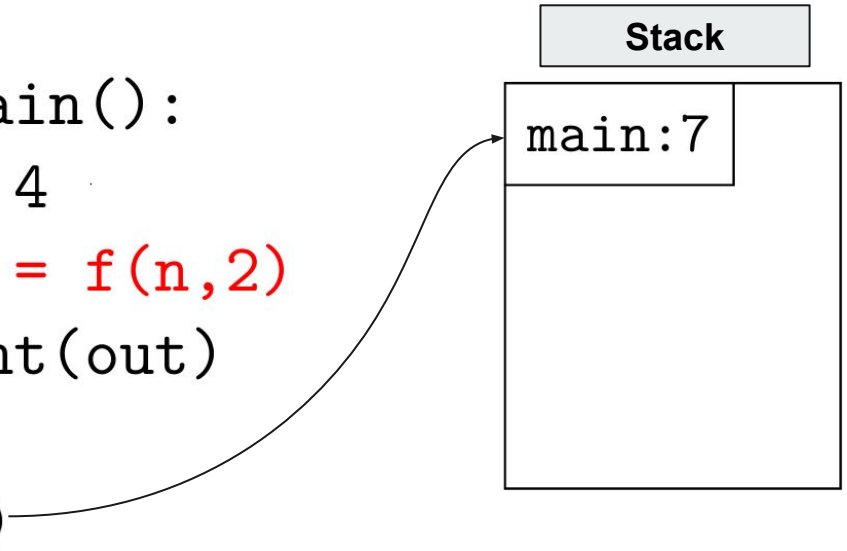
```
>>> def get_wh():
...     """Obtain weight and height from user."""
...     weight = float(input("Enter weight [pounds]: "))
...     height = float(input("Enter height [inches]: "))
...     return weight, height
...
>>> get_wh()
Enter weight [pounds]: 120
Enter height [inches]: 60
(120.0, 60.0)
>>> print(weight)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'weight' is not defined
>>> print(height)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'height' is not defined
```

Function: Print() vs Return()

```
>>> def p(x):              # p() prints a result
...     print(2 * x)
...
>>> def r(x):              # r() returns a result
...     return 2 * x
...
>>> p(10)
20
>>> r(10)
20
>>> p_result = p(10)    # Use p() in assignment operation.  Note output.
20
>>> r_result = r(10)    # Use r() in assignment operation.  No output!
>>> p_result, r_result
(None, 20)
>>> p(10) + 7
20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> r(10) + 7
27
```

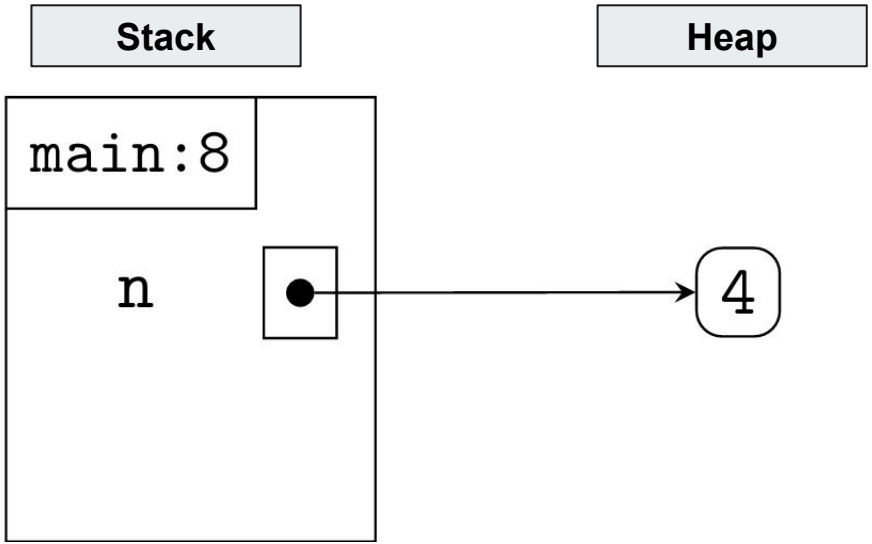
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



Function: Call Stack Frame

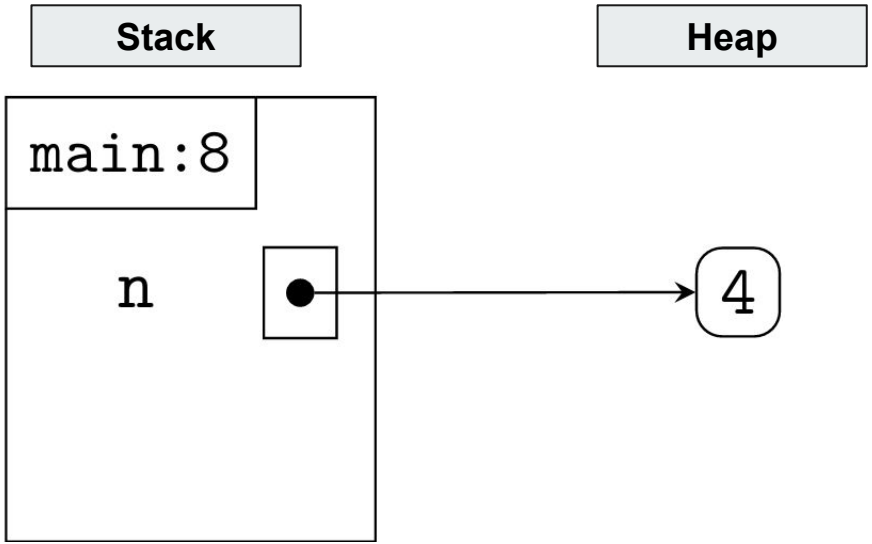
```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



Function: Call Stack Frame

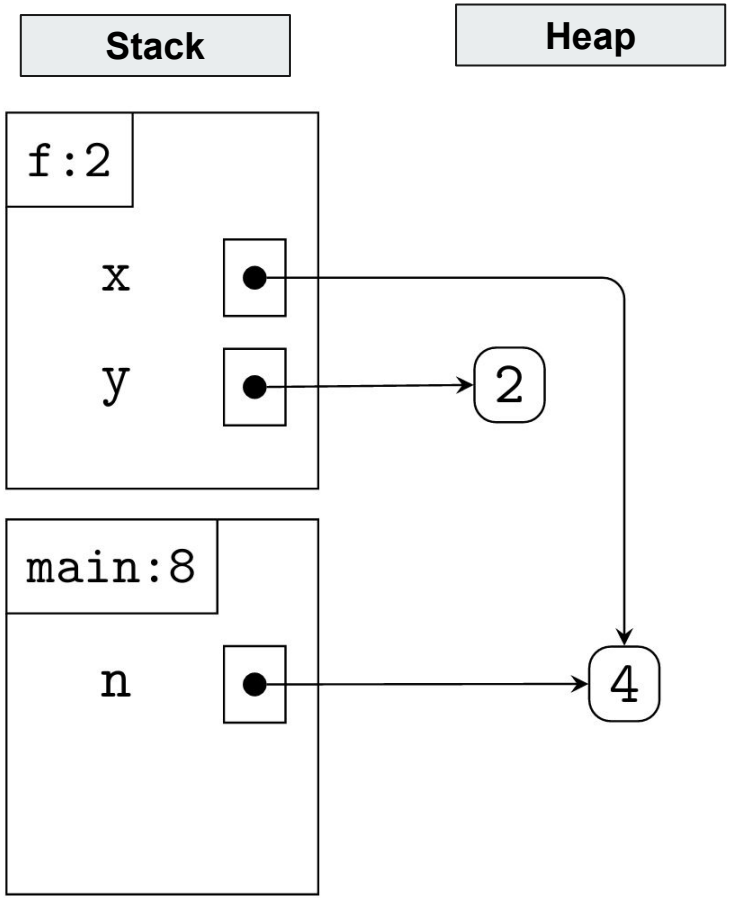
```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```

n = 4, 2; Define the argument values of *f* first



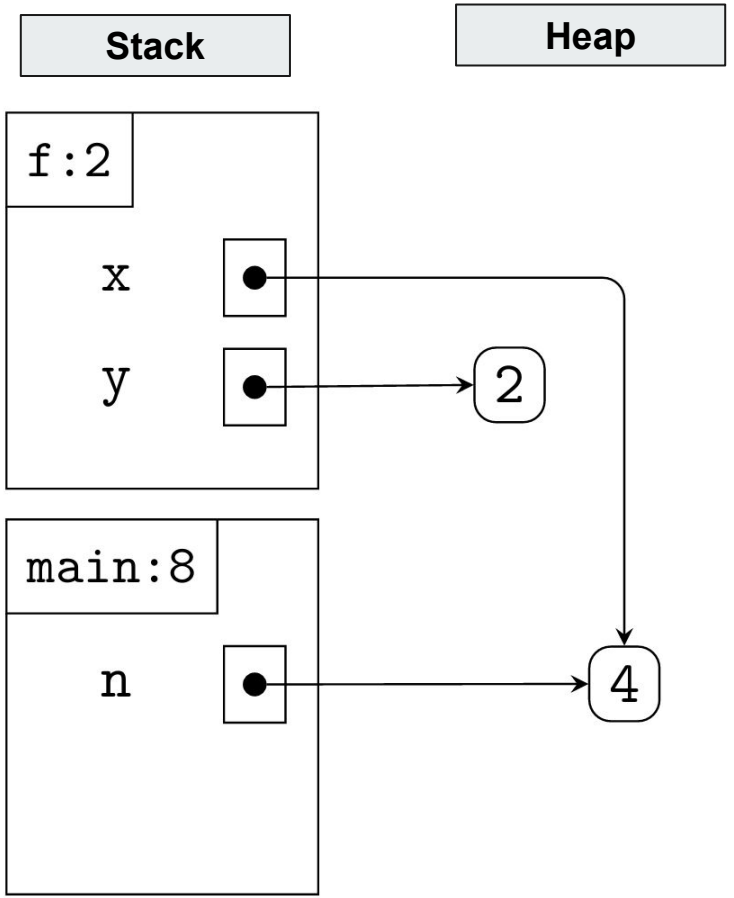
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



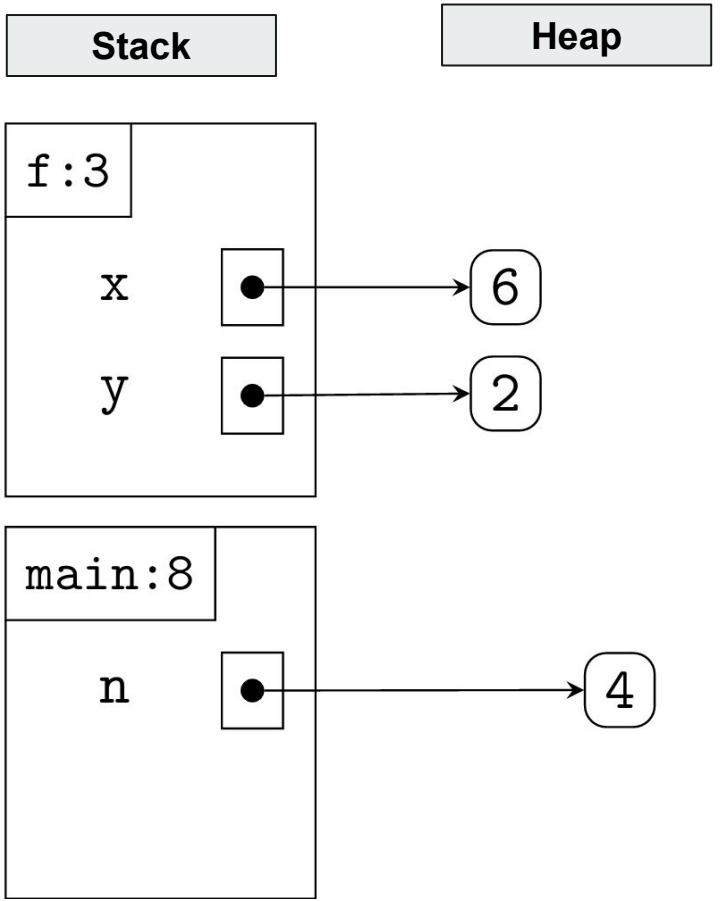
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



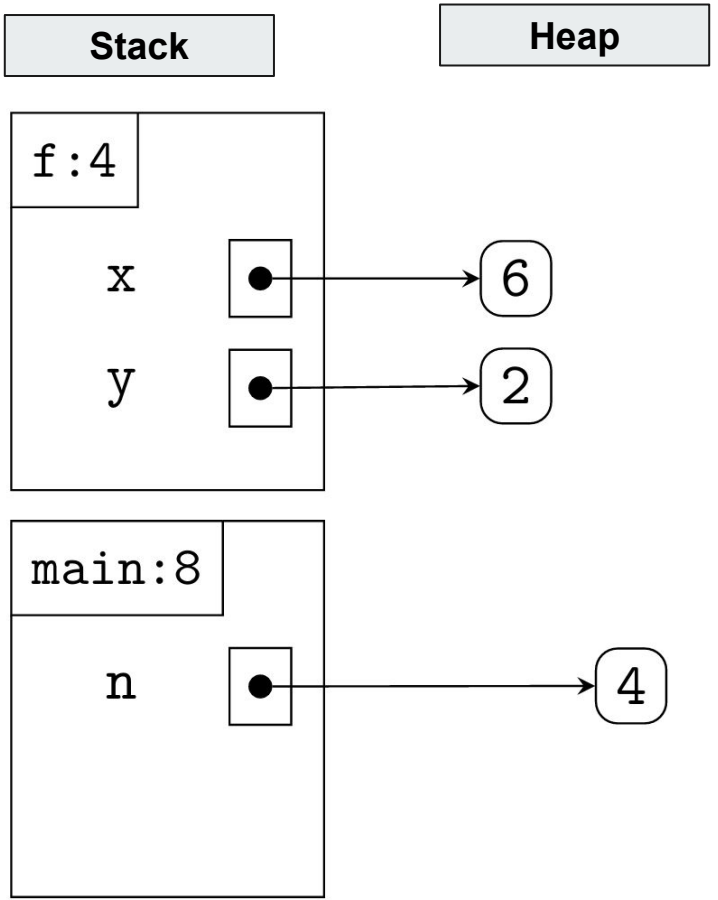
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



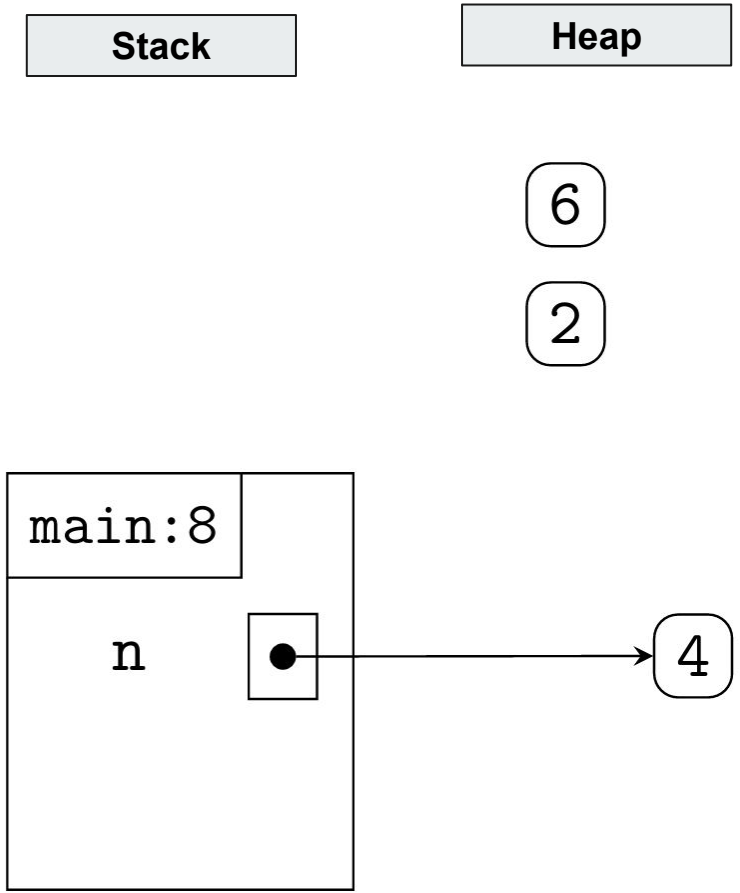
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



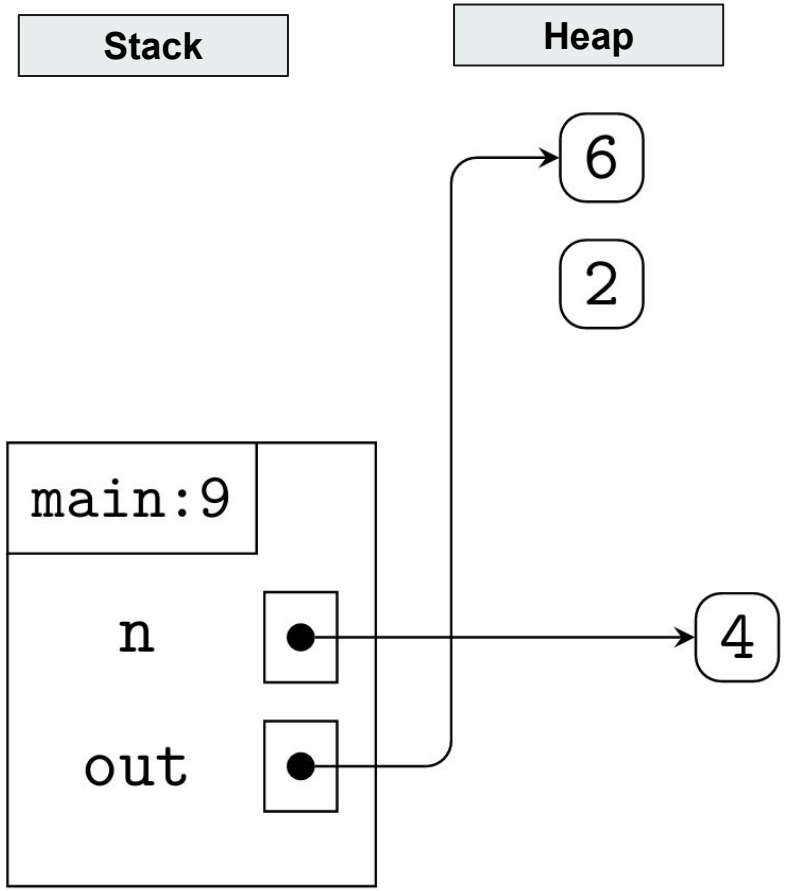
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



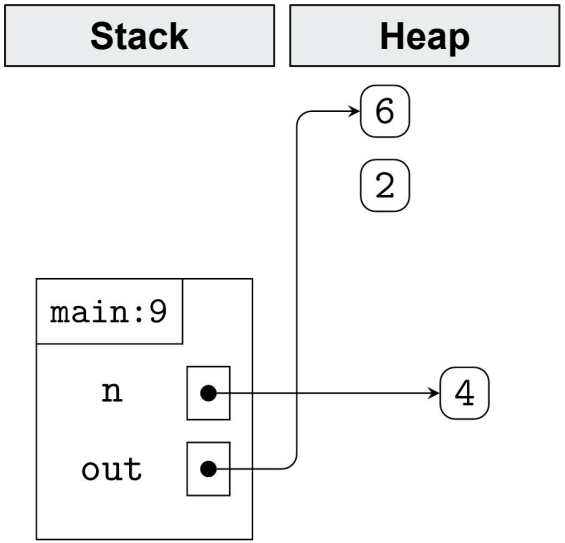
Function: Call Stack Frame

```
1  def f(x,y):
2      x = x + y
3      print(x)
4      return x
5
6  def main():
7      n = 4
8      out = f(n,2)
9      print(out)
10
11 main()
```



Function: Call Stack Frame

```
1   def f(x,y):
2       x = x + y
3       print(x)
4       return x
5
6   def main():
7       n = 4
8       out = f(n,2)
9       print(out)
10
11  main()
```



Line 9 prints the contents of the out variable (here, 6). After it runs, the main function is complete and the program is finished

String Methods

String Methods

```
>>> x = 10
>>> y = 23.4
>>> zlist = [1, 'a', [2, 3]]
>>> s0 = "Hi!"
>>> all = str(x) + str(y) + str(zlist) + str(s0)
>>> print(all)
```

Output: 1023.4[1, 'a', [2, 3]]Hi!

String Methods: Indexing

text = “abc123”

➔ How do we get the first character in a string?

text[0]

➔ How do we get the last character in a string?

text[len(text) - 1]

len() is a function used to retrieve the length of a string in the parentheses

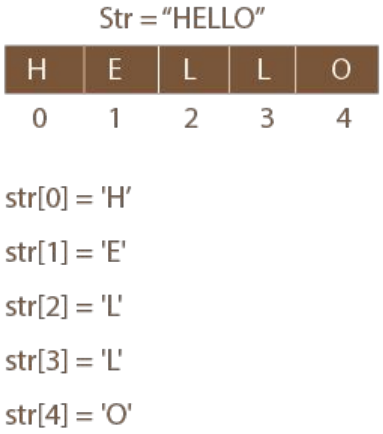
➔ What happens if we try an index outside of the string?

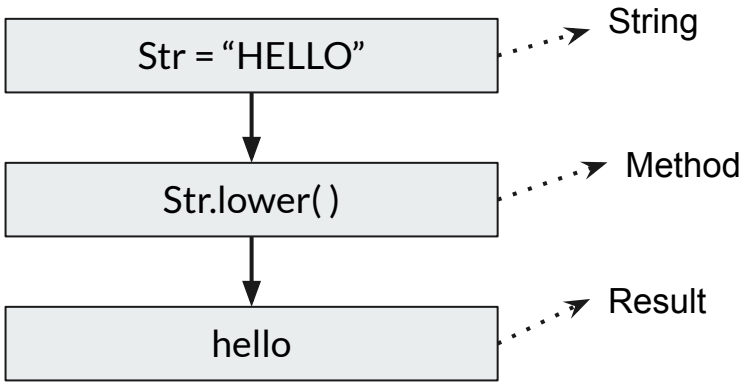
text[len(text)]

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    t[len(t)]
IndexError: string index out of range
```

String Methods

- ➔ Are similar to functions as they have parenthesis in the end
- ➔ Return a string
- ➔ Must be called along with a string
- ➔ May or may not involve arguments





String Methods: Called without string

```
>>> statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."
>>> statement
'Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'
>>> ti_statement = statement.title()
>>> print(ti_statement)
Anyone Can Learn Python, Python Is So Easy. Best Way To Learn Python Is Using It For Different Tasks.
>>> re_statement = statement.replace("python", "computer programming")
>>> print(re_statement)
Anyone can learn computer programming, computer programming is so easy. Best way to learn computer programming is using it for different tasks.
>>> print(capitalize())
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    print(capitalize())
NameError: name 'capitalize' is not defined
>>> print(title())
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print(title())
NameError: name 'title' is not defined. Did you mean: 'tuple'?
>>>
```

String Methods: Transforming Strings

str. ***lower ( )***	Returns a copy of *str* with all letters converted to lowercase.
str. ***upper ( )***	Returns a copy of *str* with all letters converted to uppercase.
str. ***capitalize ( )***	Capitalize the first character in *str* and converts the rest to lowercase.
str. ***title ( )***	Returns a copy of *str* with first character of each word capitalized.
str. ***swapcase ( )***	Returns a copy of *str* with the case of its alphabetic characters swapped.
str. ***strip ( )*** *.lstrip ( ) OR .rstrip ( )*	Removes whitespace characters from both ends of *str*.
str. ***replace (****old, new* ***)***	Returns a copy of *str* with all instances of old replaced by new.

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.upper( )	'ANYONE CAN LEARN PYTHON, PYTHON IS SO EASY. BEST WAY TO LEARN PYTHON IS USING IT FOR DIFFERENT TASKS.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.upper( )	'ANYONE CAN LEARN PYTHON, PYTHON IS SO EASY. BEST WAY TO LEARN PYTHON IS USING IT FOR DIFFERENT TASKS.'
statement.lower( )	'anyone can learn python, python is so easy. best way to learn python is using it for different tasks.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.upper( )	'ANYONE CAN LEARN PYTHON, PYTHON IS SO EASY. BEST WAY TO LEARN PYTHON IS USING IT FOR DIFFERENT TASKS.'
statement.lower( )	'anyone can learn python, python is so easy. best way to learn python is using it for different tasks.'
statement.swapcase( )	'aNYONE CAN LEARN PYTHON, PYTHON IS SO EASY. bEST WAY TO LEARN PYTHON IS USING IT FOR DIFFERENT TASKS.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.upper( )	'ANYONE CAN LEARN PYTHON, PYTHON IS SO EASY. BEST WAY TO LEARN PYTHON IS USING IT FOR DIFFERENT TASKS.'
statement.lower( )	'anyone can learn python, python is so easy. best way to learn python is using it for different tasks.'
statement.swapcase( )	'aNYONE CAN LEARN PYTHON, PYTHON IS SO EASY. bEST WAY TO LEARN PYTHON IS USING IT FOR DIFFERENT TASKS.'
statement.capitalize( )	'Anyone can learn python, python is so easy. best way to learn python is using it for different tasks.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.title( )	'Anyone Can Learn Python, Python Is So Easy. Best Way To Learn Python Is Using It For Different Tasks.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.title( )	'Anyone Can Learn Python, Python Is So Easy. Best Way To Learn Python Is Using It For Different Tasks.'
statement.strip( )	'Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

statement1= " Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks. "

Method Call	Output
statement.title( )	'Anyone Can Learn Python, Python Is So Easy. Best Way To Learn Python Is Using It For Different Tasks.'
statement1.strip( )	'Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

statement1= " Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks. "

Method Call	Output
statement.title( )	'Anyone Can Learn Python, Python Is So Easy. Best Way To Learn Python Is Using It For Different Tasks.'
statement1.strip( )	'Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'
statement1.rstrip( )	' \tAnyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'

String Methods: Transforming Strings

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

statement1= " Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks. "

Method Call	Output
statement.title( )	'Anyone Can Learn Python, Python Is So Easy. Best Way To Learn Python Is Using It For Different Tasks.'
statement1.strip( )	'Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'
statement1.rstrip( )	' \tAnyone can learn python, python is so easy. Best way to learn python is using it for different tasks.'
statement.replace('python', 'coding', 2)	'Anyone can learn coding, coding is so easy. Best way to learn python is using it for different tasks.'

String Methods: Slicing

```
>>> s1 = '"Unbroken" by Laura Hillenbrand'
>>> print(s1)
"Unbroken" by Laura Hillenbrand
>>> len(s1)        # Number of characters.
32
>>> s1[16 : 20]    # Slice of s1.
'aura'
>>> s1[1]          # Second character of s1.
'U'
>>> s1[1] = 'u'  # Attempt to change s1.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

String Methods: Slicing

- Obtaining a whole substring from a string by specifying a slice.
- Slices are exactly like ranges – they can have a *start*, an *end*, and a *step*.
- Slices are represented as numbers inside of square brackets, separated by colons. ***Slice —> Str [start : stop : step]***

Default Settings:

Start: s[0]

Stop: s[len(s)] ; *# include len(s)-1 character but exclude len(s) character*

Step: 1

s = "abcdefgh"

s[: :] or s[:] = Evaluates to s[0:len(s):1]; "abcdefgh"

s[: : -1] = Evaluates to s[len(s)-1: :-1]; "hgfedcba"

s[3:6:2] = Evaluates to "df"

String Methods: Slicing

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

```
>>> print ( statement[0:10],"<-slice1, slice2->",statement[:10])
Anyone can <-slice1, slice2-> Anyone can
>>>
```

String Methods: Slicing

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

```
>>> statement.find ( "python")
17
>>> statement[0:17 + len ( "python")  ]
'Anyone can learn python'
>>> statement[0:17 + "python"]
Traceback  ( most recent call last)  :
  File "<pyshell#11>", line 1, in <module>
    statement[0:17 + "python"]
TypeError: unsupported operand type ( s)  for +: 'int' and 'str'
>>>
```

String Methods: Slicing

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

```
>>> statement[::-1]
'.sksat tnereffid rof ti gnisu  si nohtyp nrael ot yaw tseB .ysae os si nohtyp ,nohtyp nrael nac enoynA'
>>>
```

```
>>> statement[-8:]
't tasks.'
>>>
```

```
>>> statement[-8::-1]
```

'tnereffid rof ti gnisu si nohtyp nrael ot yaw tseB .ysae os si nohtyp ,nohtyp nrael nac enoynA'

String Methods: Slicing

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

```
>>> statement[17:17+len ( "super")  ]
'pytho'
>>>
>>> statement[17:17+len ( "super")  ] = "coder"
Traceback  ( most recent call last)  :
  File "<pyshell#26>", line 1, in <module>
    statement[17:17+len ( "super")  ] = "coder"
TypeError: 'str' object does not support item assignment
>>>
```

****Strings are Immutable.****

String Methods: Immutable Strings

- Strings are ***"immutable"*** – cannot be modified
- You ***can create new objects*** that are versions of the original one
- ***Variable name*** can only be ***bound to one object***

```
s = "car"
s[0] = 'b'                    # Gives Error

s = 'b'+s[1:len(s)]           # Allowed; s is bound to new object
```

Exercise: Slicing

Suppose that you have initialized ALPHABET as

ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

so that the index numbers (in both directions) run like this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
–26	–25	–24	–23	–22	–21	–20	–19	–18	–17	–16	–15	–14	–13	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

What are the values of the following slice expressions?

1. ALPHABET[7:9]
2. ALPHABET[-3:-1]
3. ALPHABET[:3]
4. ALPHABET[-1:]
5. ALPHABET[14:-12]
5. ALPHABET[1:-1]
6. ALPHABET[0:5:2]
7. ALPHABET[::-1]
8. ALPHABET[5:2:-1]
9. ALPHABET[14:2:-3]

String Methods: Finding Patterns and Index

**Search is case-sensitive*

str. ***fnd (pattern)***	Returns the first index of *pattern* in *str* or -1 if it does not appear.
str. ***find (pattern , k)***	Same as above but start searching the *pattern* with index *k*.
str. ***rfind(pattern)*** *str.* ***rfind(pattern, k)***	Returns the last index of *pattern* in *str* or -1 if it does not appear. Same as above but start searches backward the *pattern* from index *k*.
str. ***startswith (prefix)***	Returns **TRUE** if the *str* starts with the *prefix*.
str. ***endswith (suffix)***	Returns **TRUE** if the *str* ends with the *suffix*.
str. ***index (pattern)***	Returns the first index of *pattern* in *str* or *ValueError* if it does not appear.
str. ***__repr__ ( )***	Returns a detailed string representation of object in *str* ; for debugging

String Methods: Finding Patterns and Index

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.find(‘python’)	17

String Methods: Finding Patterns and Index

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.find(‘python’)	17
statement.index(‘python’)	17

String Methods: Finding Patterns and Index

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.find(‘python’)	17
statement.index(‘python’)	17
statement.count(‘python’)	3

String Methods: split() & join()

statement = "Anyone can learn python, python is so easy. Best way to learn python is using it for different tasks."

Method Call	Output
statement.split( ) If no argument is given, the splitting is done at every whitespace occurrence	[‘Anyone’, ‘can’, ‘learn’, ‘python’, ‘,’, ‘ python’, ‘is’, ‘so’, ‘easy’, ‘.’, ‘Best’, ‘way’, ‘to’, ‘learn’, ‘python’, ‘is’, ‘using’, ‘it’, ‘for’, ‘different’, ‘tasks.’]
“ ”.join([“smashed”, “together”])	‘smashed together’
“-”.join([“one”, “by”, “one”])	‘one-by-one’

String Methods: Format Specifiers

ch`.isalpha()` Returns `True` if *ch* is a letter.
ch`.isdigit()` Returns `True` if *ch* is a digit.
ch`.isalnum()` Returns `True` if *ch* is a letter or a digit.
ch`.islower()` Returns `True` if *ch* is a lowercase letter.
ch`.isupper()` Returns `True` if *ch* is an uppercase letter.
ch`.isspace()` Returns `True` if *ch* is a ***whitespace character*** (space, tab, or newline).
str`.isidentifier()` Returns `True` if this string is a legal Python identifier.

String Methods: Membership

```
>>> "a" in "abc"
True
>>> "A" in "abc"
False

>>> "ab" in "abc"
True
>>> "bc" in "abc"
True
>>> "bc" not in "abc"
False
```

String Methods: Membership

```
>>> x = "cat" < "rat"
```

String Methods: Membership

```
>>> x = "cat" < "rat"
>>> print(x, type(x))
```

String Methods: Membership

```
>>> x = "cat" < "rat"
>>> print(x, type(x))
True <class 'bool'>
```

String Methods: Membership

```
>>> x = "cat" < "rat"
>>> print(x, type(x))
True <class 'bool'>

>>> myString = "12321"
```

String Methods: Membership

```
>>> x = "cat" < "rat"
>>> print(x, type(x))
True <class 'bool'>
```

```
>>> myString = "12321"
>>> myString == myString[::-1]
```

String Methods: Membership

```
>>> x = "cat" < "rat"
>>> print(x, type(x))
True <class 'bool'>
```

```
>>> myString = "12321"
>>> myString == myString[::-1]
True
```

Function: Scope

Scope: Role of Functions

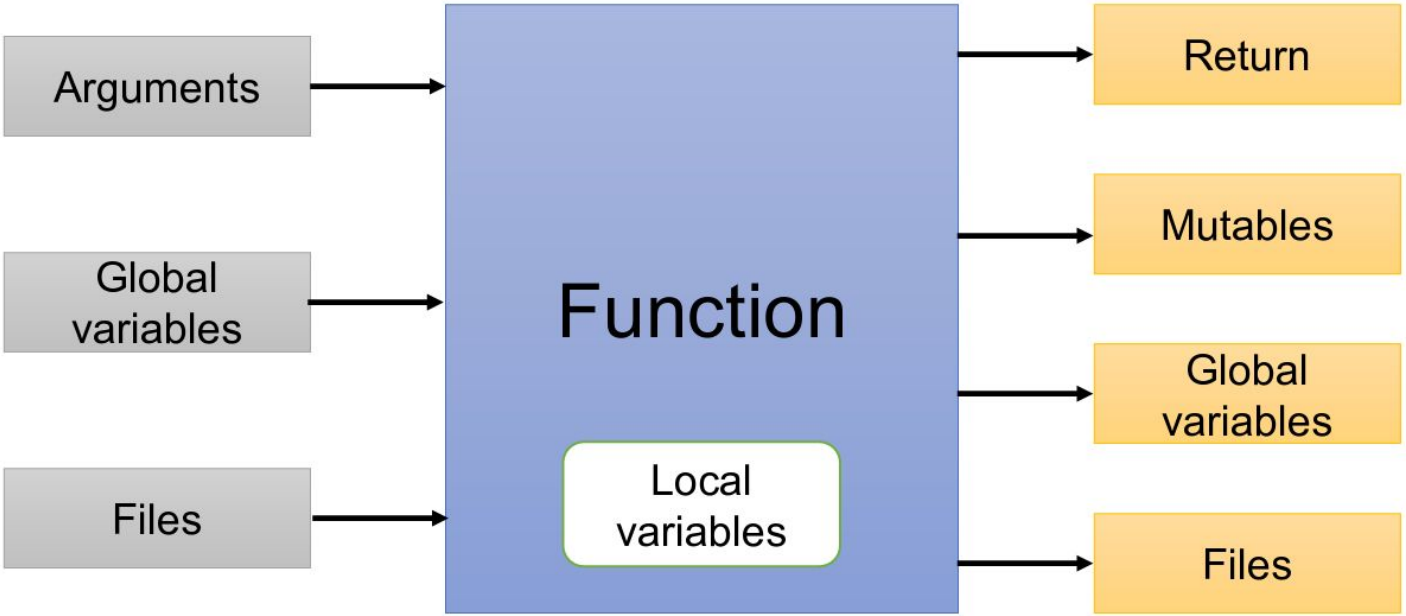


Image Credit: Learning Python by Mark Lutz

Function: Namespace

Scope: The region in the program code where variables and names are accessible.

- Namespace is the complete list of names, including all objects, functions, etc. that exist in a given context.
- Whenever we use a name, such as a variable or a function name, Python searches through different scope levels (or namespaces) to determine whether the name exists or not.
    - Built-in Namespace: functions or exceptions that have already built
    - Global Namespace: Available through main program
    - Local Namespace: Within the function

Scope: Accessibility

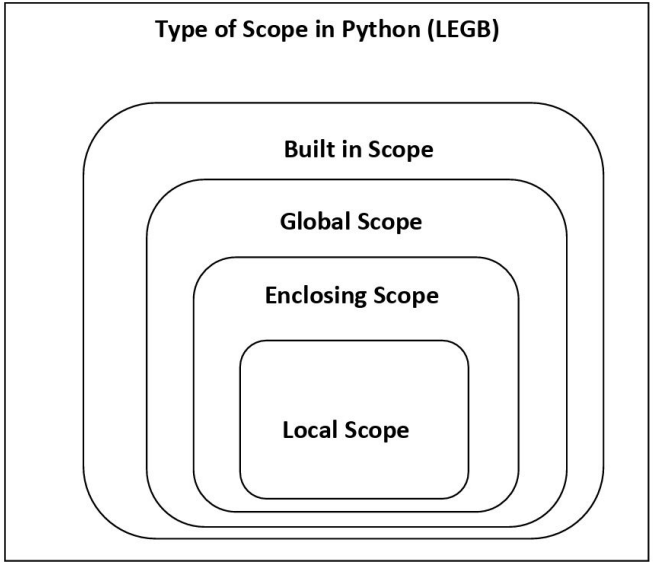
Scope concept follows the ***LEGB*** (*Local*, *Enclosing*, *Global* and *built-in*) rule.

- The scope of an object is the ***namespace*** within which the object is available.
- Access depends on where the name is defined

Scope: LEGB Rule

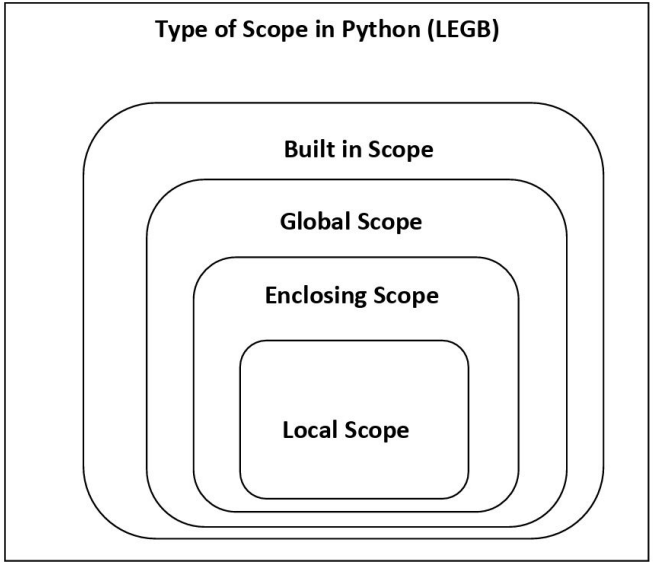
Scope concept follows the ***LEGB*** (*Local*, *Enclosing*, *Global* and *built-in*) rule.

- The scope of an object is the ***namespace*** within which the object is available.
- Access depends on where the name is defined



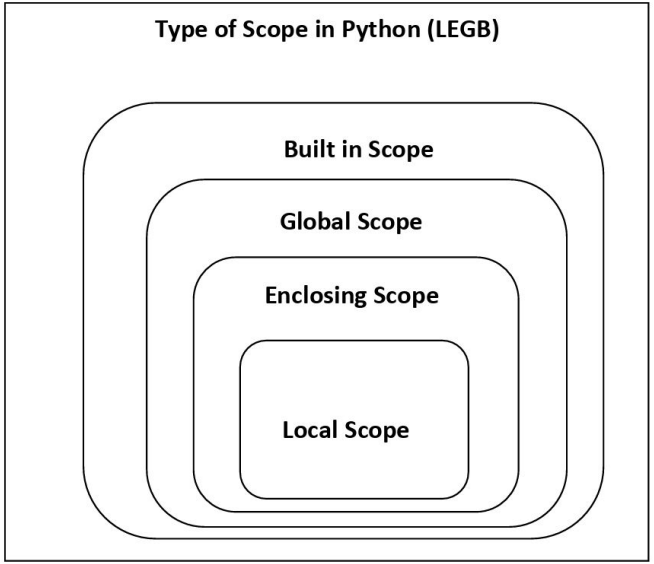
Scope: LEGB Rule

- Local first: Look for this name first in the local namespace and use local version if available. If not, go to higher namespace.



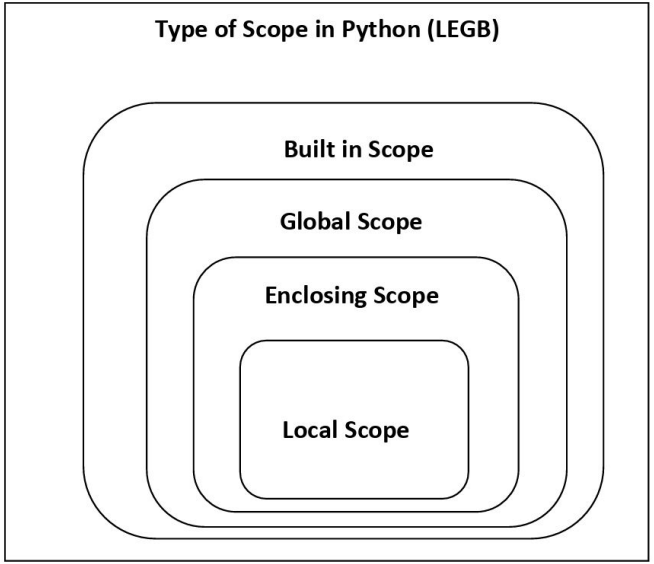
Scope: LEGB Rule

- **E**nclosing second: If the current function is enclosed within another function (nested functions), look for the name in that outer function. If not, go to higher namespace.



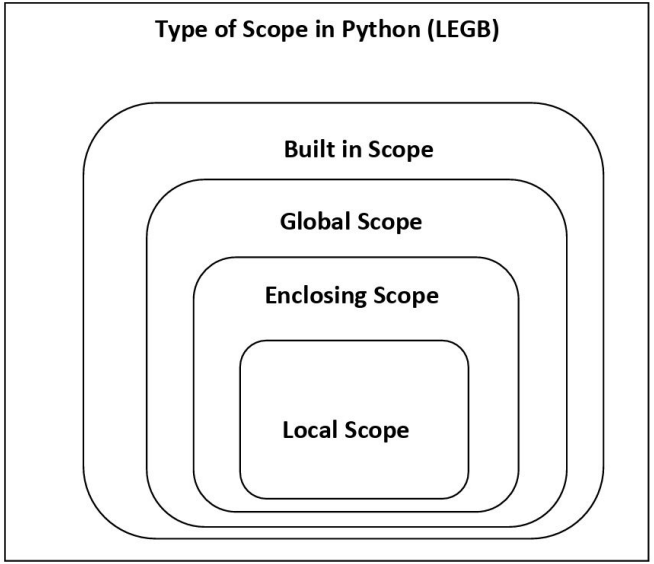
Scope: LEGB Rule

- **G**lobal third: Look for the name in the objects defined at global scale (e.g., main program).



Scope: LEGB Rule

- **B**uilt-in last: Finally, look for the variable among Python's built-in names.



LEGB Rule: Local Scope

```
#Script to understand Local Scope:

#name "base" is used for input variable
def square(base):
        return  base ** 2
```

LEGB Rule: Local Scope

```
#Script to understand Local Scope:

#name "base" is used for input variable
def square(base):
        return  base ** 2

#name "base" is reused for input variable
def cube(base):
        return  base ** 3
```

LEGB Rule: Local Scope

```
#Script to understand Local Scope:

#name "base" is used for input variable
def square(base):
        return  base ** 2

#name "base" is reused for input variable
def cube(base):
        return  base ** 3

inp = 8
print(square(inp))
print(cube(inp))
#no confusion which base is being referred to!
```

LEGB Rule: Local Scope

```
#Script to understand Local Scope:

#name "base" is used for input variable
def square(base):
        return  base ** 2

#name "base" is reused for input variable
def cube(base):
        return  base ** 3

inp = 8
print(square(inp))
print(cube(inp))
#no confusion which base is being referred to!
```

Output

```
64
512
```

LEGB Rule: Enclosing or Non-Local Scope

```
def outer():
    var = 100
```

LEGB Rule: Enclosing or Non-Local Scope

```
def outer():
    var = 100
    def inner():
        print('Printing var from inner function:', var)
```

LEGB Rule: Enclosing or Non-Local Scope

```
def outer():
    var = 100
    def inner():
        print('Printing var from inner function:', var)
    inner()
```

LEGB Rule: Enclosing or Non-Local Scope

```
def outer():
    var = 100
    def inner():
        print('Printing var from inner function:', var)
    inner()
    print('Printing var from outer function:', var)
```

LEGB Rule: Enclosing or Non-Local Scope

```
def outer():
    var = 100
    def inner():
        print('Printing var from inner function:', var)
    inner()
    print('Printing var from outer function:', var)

outer()
```

```
Printing var from inner function: 100
Printing var from outer function: 100
```

Local scope of outer ( ) is the enclosing scope of inner ( )

LEGB Rule: Enclosing or Non-Local Scope

```
def outer():
    var = 100
    def inner():
        var = 200
        print('Printing var from inner function:', var)
    inner()
    print('Printing var from outer function:', var)

outer()
```

```
Printing var from inner function: 200
Printing var from outer function: 100
```

Local scope of outer ( ) is the enclosing scope of inner ( )

LEGB Rule: Global Scope

```
# function f calls g, which inturn
# calls h without arguemnts, main calls f
def f():
    x = 5
    g()
    print('in f', x)
```

LEGB Rule: Global Scope

```
# function f calls g, which inturn
# calls h without arguemnts, main calls f
def f():
    x = 5
    g()
    print('in f', x)

def g():
    x = 7
    h()
```

LEGB Rule: Global Scope

```
# function f calls g, which inturn
# calls h without arguemnts, main calls f
def f():
    x = 5
    g()
    print('in f', x)

def g():
    x = 7
    h()

def h():
    print('in h', x)
```

LEGB Rule: Global Scope

```
# function f calls g, which inturn
# calls h without arguemnts, main calls f
def f():
    x = 5
    g()
    print('in f', x)

def g():
    x = 7
    h()

def h():
    print('in h', x)

x = 3
f()
print('in main', x)
```

LEGB Rule: Global Scope

```
# function f calls g, which inturn
# calls h without arguemnts, main calls f
def f():
    x = 5
    g()
    print('in f', x)

def g():
    x = 7
    h()

def h():
    print('in h', x)

x = 3
f()
print('in main', x)
```

Output

```
in h 3
in f 5
in main 3
```

LEGB Rule: Global Scope

```
def outer():
    #defines local scope of outer()
    #als defines enclosing scope of inner()
    def inner():
        print(number)
    inner()

number = 10
outer()
```

Output 10

LEGB Rule: Built-In Scope

```
>>> myList = [10, 2, -3]
>>> print(sum(myList))
```

LEGB Rule: Built-In Scope

```
>>> myList = [10, 2, -3]
>>> print(sum(myList))
9
```

LEGB Rule: Built-In Scope

```
>>> myList = [10, 2, -3]
>>> print(sum(myList))
9
>>> sum = 3
```

LEGB Rule: Built-In Scope

```
>>> myList = [10, 2, -3]
>>> print(sum(myList))
9
>>> sum = 3
>>> newList = [2, 3, 4]
```

LEGB Rule: Built-In Scope

```
>>> myList = [10, 2, -3]
>>> print(sum(myList))
9
>>> sum = 3
>>> newList = [2, 3, 4]
>>> print(sum(newList))
```

LEGB Rule: Built-In Scope

```
>>> myList = [10, 2, -3]
>>> print(sum(myList))
9
>>> sum = 3
>>> newList = [2, 3, 4]
>>> print(sum(newList))
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(sum(newList))
TypeError: 'int' object is not callable
```

Remember: ***do not redefine built-in names!***

Function: Arguments

Function: Argument Passing

- The objects sent to a function are called its arguments. Arguments are sometimes also called parameters.
- Passing an object as a argument to a function passes a reference to that object.
- Argument names in the def line of the function become new, local names.
- Arguments are passed in two ways:
    - ‘Immutables’ are passed by value eg., String, Integer, Tuple
    - ‘Mutables’ are passed by reference eg., Lists

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName

#In the following function, 2nd argument is optional
#It has default value 'Gupta', which we can omit in its call
def concatNameV3(firstName, lastName = 'Gupta'):
    return firstName + ' ' + lastName
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName

#In the following function, 2nd argument is optional
#It has default value 'Gupta', which we can omit in its call
def concatNameV3(firstName, lastName = 'Gupta'):
    return firstName + ' ' + lastName


#main program:
print(concatNameV1('Ajay', 'Gupta'))
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName

#In the following function, 2nd argument is optional
#It has default value 'Gupta', which we can omit in its call
def concatNameV3(firstName, lastName = 'Gupta'):
    return firstName + ' ' + lastName


#main program:
print(concatNameV1('Ajay', 'Gupta'))
print(concatNameV2('Kumari', 'Preeti'))
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName

#In the following function, 2nd argument is optional
#It has default value 'Gupta', which we can omit in its call
def concatNameV3(firstName, lastName = 'Gupta'):
    return firstName + ' ' + lastName


#main program:
print(concatNameV1('Ajay', 'Gupta'))
print(concatNameV2('Kumari', 'Preeti'))
print(concatNameV3('Rahul', 'Nair'))
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName

#In the following function, 2nd argument is optional
#It has default value 'Gupta', which we can omit in its call
def concatNameV3(firstName, lastName = 'Gupta'):
    return firstName + ' ' + lastName


#main program:
print(concatNameV1('Ajay', 'Gupta'))
print(concatNameV2('Kumari', 'Preeti'))
print(concatNameV3('Rahul', 'Nair'))
print(concatNameV3('Ajay'))
```

Function: Arguments

```
#function definitions:
def concatNameV1(firstName, lastName):
    return firstName + ' ' + lastName

def concatNameV2(lastName, firstName):
    return firstName + ' ' + lastName

#In the following function, 2nd argument is optional
#It has default value 'Gupta', which we can omit in its call
def concatNameV3(firstName, lastName = 'Gupta'):
    return firstName + ' ' + lastName


#main program:
print(concatNameV1('Ajay', 'Gupta'))
print(concatNameV2('Kumari', 'Preeti'))
print(concatNameV3('Rahul', 'Nair'))
print(concatNameV3('Ajay'))
```

Output

```
Ajay Gupta
Preeti Kumari
Rahul Nair
Ajay Gupta
```

Function: Arguments

```
#function definition:
def appendEnd(myList = []):
    myList.append('end')
    return myList
```

Function: Arguments

```
#function definition:
def appendEnd(myList = []):
    myList.append('end')
    return myList

#main program:
a = ['Monday']
appendEnd(a)
```

Function: Arguments

```
#function definition:
def appendEnd(myList = []):
    myList.append('end')
    return myList

#main program:
a = ['Monday']
appendEnd(a)
print(a)#['Monday', 'end']
```

Function: Arguments

```
#function definition:
def appendEnd(myList = []):
    myList.append('end')
    return myList

#main program:
a = ['Monday']
appendEnd(a)
print(a)#['Monday', 'end']

appendEnd(a)
print(a) #['Monday', 'end', 'end']
```

Function: Arguments

```
#function definition:
def appendEnd(myList = []):
    myList.append('end')
    return myList

#main program:
a = ['Monday']
appendEnd(a)
print(a)#['Monday', 'end']

appendEnd(a)
print(a) #['Monday', 'end', 'end']

#Unexpected behavior for mutable objects.
#Hence, Avoid it:-
b = appendEnd()
b = appendEnd()
```

Function: Arguments

```
#function definition:
def appendEnd(myList = []):
    myList.append('end')
    return myList

#main program:
a = ['Monday']
appendEnd(a)
print(a)#['Monday', 'end']

appendEnd(a)
print(a) #['Monday', 'end', 'end']

#Unexpected behavior for mutable objects.
#Hence, Avoid it:-
b = appendEnd()
b = appendEnd()
print(b) #['end', 'end']
```

Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denominator was 0.")

print(calc(add, 2, 3))
```

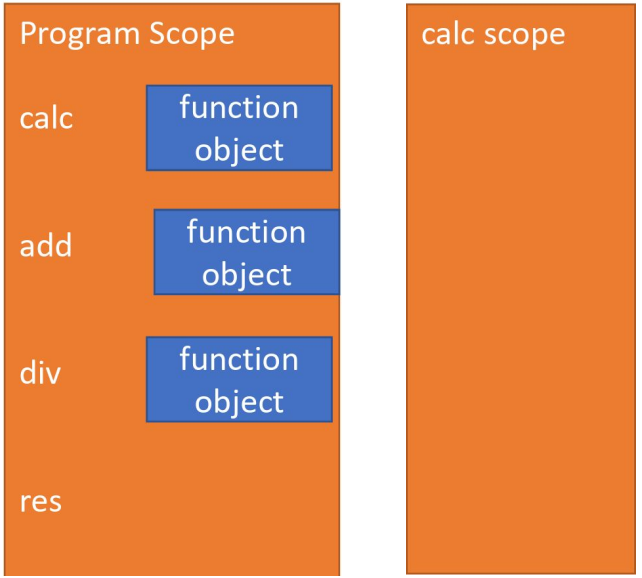
Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



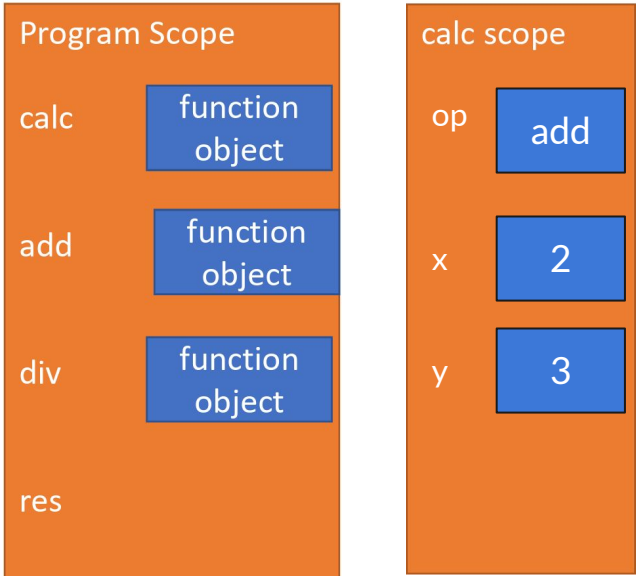
Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



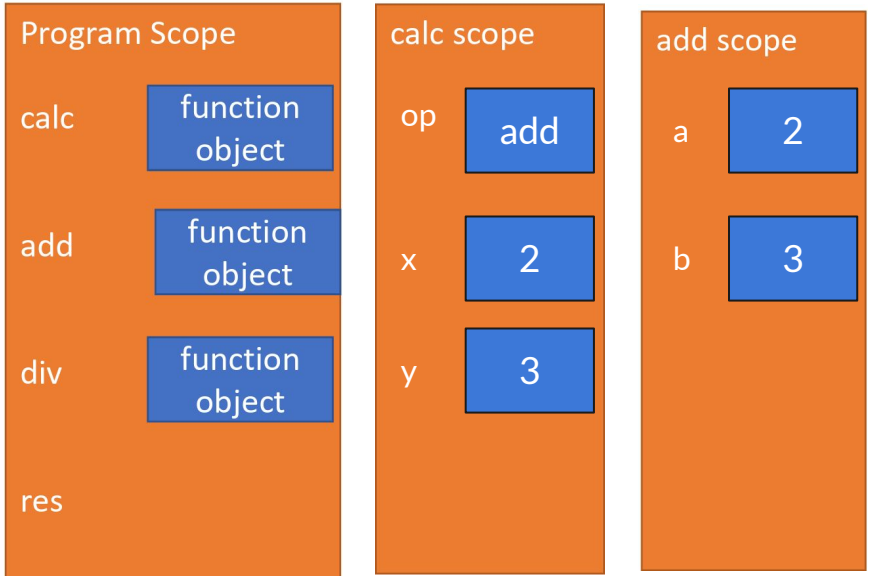
Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



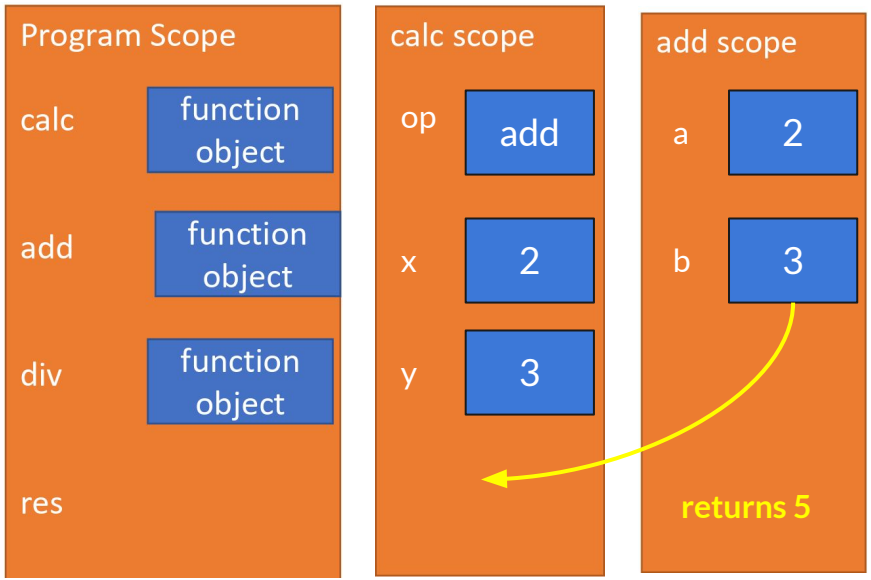
Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



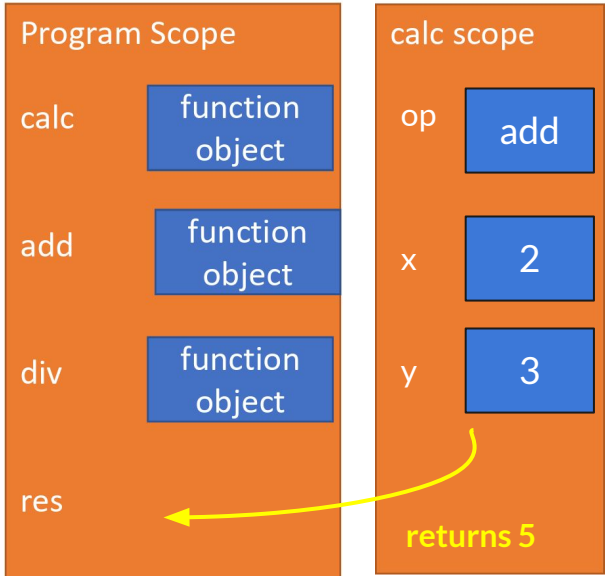
Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



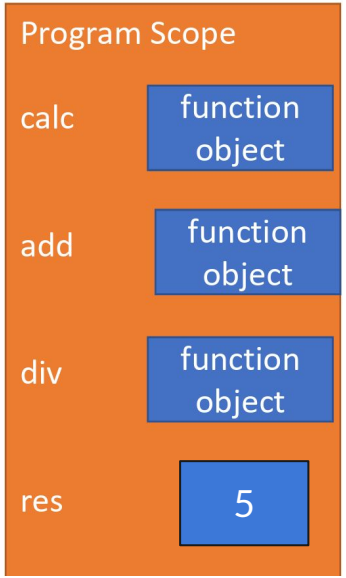
Function: Arguments

```
def calc(op, x, y):
    return op(x,y)

def add(a,b):
    return a+b

def div(a,b):
    if b != 0:
        return a/b
    print("Denom was 0.")

res = calc(add, 2, 3)
```



Function As Arguments: Questions

```
#function definition:
def f(y):
    y = 10

#main program:
x = 5

print(x)
#prints 5

f(x)

print(x)
#prints 5
```

Function As Arguments: Questions

```
#functions definition:-
def swapV1(a,b):
    temp = a
    a = b
    b = temp

def swapV2(a,b):
    return b, a # returns more than one value

#main program:-
i, j = 2, 3
print("Original i, j:", i, j)

swapV1(i,j)
print("After swapV1:", i, j)

i, j = swapV2(i,j)
print("After swapV2:", i, j)
```

```
Original i, j: 2 3
After swapV1: 2 3
After swapV2: 3 2
```