Python Introduction

# Where is Python used?

Python is a versatile programming language used in a wide range of applications, from game development to scientific and numeric computing, web development, desktop GUIs, and software development. Major companies and services such as Dropbox, UBER, Spotify, Pinterest, and BuzzFeed use Python extensively in their technology stacks. Python is also popular in education and is often used to teach programming due to its simplicity and ease of use. It's an excellent choice for building web applications, scientific computing, desktop GUIs, and business applications.

# Compiler vs Interpreter

Two ways of translating high-level programming language to machine language: compilation and interpretation.

Compilation translates source code once and distributes as machine code file, while interpretation translates source code each time it is run.

Most programming languages are designed to be either compiled or interpreted, and not both.

A high-level programming language is usually designed to fall into one of these two categories depending on whether it is intended to be compiled or interpreted.

## Interpreter

When a computer program is written, it exists as a text file that contains the source code. The source code must be in plain text format without any decorations. To execute the program, an interpreter is used to read and process the source code line by line from top to bottom and left to right. The interpreter checks if all subsequent lines are correct and if it finds an error, it immediately stops and displays an error message indicating the location and cause of the error. However, the error message may be misleading and not always located at the actual cause of the error. If the line is correct, the interpreter tries to execute it, and the "read-check-execute" trio can be repeated multiple times. It is possible for a significant part of the code to be executed successfully before an error is detected. There is no clear answer as to which method, compiling or interpreting, is better as both have their advantages and disadvantages.

COMPILATION

the execution of the translated code is usually faster; only the user has to have the compiler - the end-user may use the code without it;

the translated code is stored using machine language - as it is very hard to understand it, your own inventions and programming tricks are likely to remain your secret.

the compilation itself may be a very time-consuming process - you may not be able to run your code immediately after making an amendment;

you have to have as many compilers as hardware platforms you want your code to be run on.

INTERPRETER

you can run the code as soon as you complete it - there are no additional phases of translation;the code is stored using programming language, not machine language - this means that it can be run on computers using different machine languages; you don't compile your code separately for each different architecture. don't expect interpretation to ramp up your code to high speed - your code will share the computer's power with the interpreter, so it can't be really fast both you and the end user have to have the interpreter to run your code.

# Introduction

* Python is a high-level programming language with dynamic semantics, widely used for general-purpose programming.
* The name of the Python programming language comes from the BBC television comedy sketch series called Monty Python's Flying Circus.
* Python was created by Guido van Rossum, born in 1956 in Haarlem, the Netherlands.
* Python is one of the few languages whose authors are known by name.
* The speed with which Python has spread around the world is a result of the continuous work of thousands of programmers, testers, users, and enthusiasts.

Reasons to use Python:

* Easy to learn, allowing for faster programming
* Easy to teach, allowing for more focus on general programming techniques
* Allows for faster writing of new software
* Code is often easier to understand, both for writing and reading
* Free, open, and multiplatform

Drawbacks of Python:

* Not the fastest programming language
* May be resistant to simpler testing techniques, making debugging more difficult
* However, making mistakes is also harder in Python.

# Types of Python

1. CPython: The standard implementation of Python written in C.
2. Cython: automatically translates the Python code (clean and clear, but not too swift) into "C" code (complicated and talkative, but agile).
3. Jython: A Python implementation written in Java and designed to run on the Java Virtual Machine (JVM).
4. IronPython: An implementation of Python for .NET and Mono platforms.
5. PyPy: A fast, compliant alternative implementation of Python that includes a Just-In-Time (JIT) compiler. a Python within a Python. In other words, it represents a Python environment written in Python-like language named RPython (Restricted Python). It is actually a subset of Python.
6. The source code of PyPy is not run in the interpretation manner, but is instead translated into the C programming language and then executed separately.
7. MicroPython: A lightweight implementation of Python that is designed to run on microcontrollers and other small devices.
8. Anaconda: A popular distribution of Python used for data science and scientific computing that includes a number of scientific libraries and tools.

# Print() function

1. The print() function is a **built-in** function. It prints/outputs a specified message to the screen/console window.

2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.8 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the [Python Standard Library](https://docs.python.org/3/library/functions.html).

3. To call a function (this process is known as **function invocation** or **function call**), you need to use the function name followed by parentheses. You can pass arguments into a function by placing them inside the parentheses. You must separate arguments with a comma, e.g., print("Hello,", "world!"). An "empty" print() function outputs an empty line to the screen.

4. Python strings are delimited with **quotes**, e.g., "I am a string" (double quotes), or 'I am a string, too' (single quotes).

5. Computer programs are collections of **instructions**. An instruction is a command to perform a specific task when executed, e.g., to print a certain message to the screen.

6. In Python strings the **backslash** (\) is a special character which announces that the next character has a different meaning, e.g., \n (the **newline character**) starts a new output line.

7. **Positional arguments** are the ones whose meaning is dictated by their position, e.g., the second argument is outputted after the first, the third is outputted after the second, etc.

8. **Keyword arguments** are the ones whose meaning is not dictated by their location, but by a special word (keyword) used to identify them.

9. The end and sep parameters can be used for formatting the output of the print() function. The sep parameter specifies the separator between the outputted arguments, e.g., print("H", "E", "L", "L", "O", sep="-"), whereas the end parameter specifies what to print at the end of the print statement.

# Function Invocation

* A function invocation is a statement that tells Python to execute a function with a specific name and arguments.
* The function invocation consists of the function name followed by parentheses that contain the argument(s) passed to the function. For example, in the statement print("Hello, World!"), "print" is the function name, and "Hello, World!" is the argument passed to the function.
* When Python encounters a function invocation, it checks if the function name is valid and if the number of arguments passed to the function matches the function's requirements. If either condition fails, Python will abort the code and raise an error.
* If the function name and arguments are valid, Python will execute the code inside the function and perform any desired effects or evaluations. For example, the print() function will output the argument(s) to the console.
* After the function finishes executing, Python returns to the point in the code where the function was invoked and continues executing the remaining code.
* Functions are an essential part of Python programming as they allow you to break down complex tasks into smaller, more manageable parts that can be reused and combined in various ways.

# Literals

1. **Literals** are notations for representing some fixed values in code. Python has various types of literals - for example, a literal can be a number (numeric literals, e.g., 123), or a string (string literals, e.g., "I am a literal.").

2. The **binary system** is a system of numbers that employs *2* as the base. Therefore, a binary number is made up of 0s and 1s only, e.g., 1010 is *10* in decimal.

Octal and hexadecimal numeration systems, similarly, employ *8* and *16* as their bases respectively. The hexadecimal system uses the decimal numbers and six extra letters.

3. **Integers** (or simply **int**s) are one of the numerical types supported by Python. They are numbers written without a fractional component, e.g., 256, or -1 (negative integers). Python allows underscore Therefore, you can write this number either like this: 11111111, or like that: 11\_111\_111.

If an integer number is preceded by an 0O or 0o prefix (zero-o), it will be treated as an octal value. This means that the number must contain digits taken from the [0..7] range only.

0o123 is an octal number with a (decimal) value equal to 83.

The second convention allows us to use hexadecimal numbers. Such numbers should be preceded by the prefix 0x or 0X (zero-x).

0x123 is a hexadecimal number with a (decimal) value equal to 291.

4. **Floating-point** numbers (or simply **float**s) are another one of the numerical types supported by Python. They are numbers that contain (or are able to contain) a fractional component, e.g., 1.27.

5. To encode an apostrophe or a quote inside a string you can either use the escape character, e.g., 'I\'m happy.', or open and close the string using an opposite set of symbols to the ones you wish to encode, e.g., "I'm happy." to encode an apostrophe, and 'He said "Python", not "typhoon"' to encode a (double) quote.

6. **Boolean values** are the two constant objects True and False used to represent truth values (in numeric contexts 1 is True, while 0 is False.

**EXTRA**

There is one more, special literal that is used in Python: the None literal. This literal is a so-called NoneType object, and it is used to represent **the absence of a value**.

# Operators and Expression

1. An **expression** is a combination of values (or variables, operators, calls to functions ‒ you will learn about them soon) which evaluates to a certain value, e.g., 1 + 2.

2. **Operators** are special symbols or keywords which are able to operate on the values and perform (mathematical) operations, e.g., the \* operator multiplies two values: x \* y.

3. Arithmetic operators in Python: + (addition), - (subtraction), \* (multiplication), / (classic division ‒ always returns a float), % (modulus ‒ divides left operand by right operand and returns the remainder of the operation, e.g., 5 % 2 = 1), \*\* (exponentiation ‒ left operand raised to the power of right operand, e.g., 2 \*\* 3 = 2 \* 2 \* 2 = 8), // (floor/integer division ‒ returns a number resulting from division, but rounded down to the nearest whole number, e.g., 3 // 2.0 = 1.0)

4. A **unary** operator is an operator with only one operand, e.g., -1, or +3.

5. A **binary** operator is an operator with two operands, e.g., 4 + 5, or 12 % 5.

6. Some operators act before others – **the hierarchy of priorities**:

* the \*\* operator (exponentiation) has the highest priority;
* then the unary + and - (note: a unary operator to the right of the exponentiation operator binds more strongly, for example: 4 \*\* -1 equals 0.25)
* then \*, /, //, and %;
* and, finally, the lowest priority: the binary + and -.

7. Subexpressions in **parentheses** are always calculated first, e.g., 15 - 1 \* (5 \* (1 + 2)) = 0.

8. The **exponentiation** operator uses **right-sided binding**, e.g., 2 \*\* 2 \*\* 3 = 256.

# Variables

1. A **variable** is a named location reserved to store values in the memory. A variable is created or initialized automatically when you assign a value to it for the first time. (2.1.4.1)

2. Each variable must have a unique name - an **identifier**. A legal identifier name must be a non-empty sequence of characters, must begin with the underscore(\_), or a letter, and it cannot be a Python keyword. The first character may be followed by underscores, letters, and digits. Identifiers in Python are case-sensitive. (2.1.4.1)

3. Python is a **dynamically-typed** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (=) sign, i.e., var = 1.

4. You can also use **compound assignment operators** (shortcut operators) to modify values assigned to variables, e.g., var += 1, or var /= 5 \* 2.

5. You can assign new values to already existing variables using the assignment operator or one of the compound operators,

# Comments

1. Comments can be used to leave additional information in code. They are omitted at runtime. The information left in source code is addressed to human readers. In Python, a comment is a piece of text that begins with #. The comment extends to the end of line.

2. If you want to place a comment that spans several lines, you need to place # in front of them all. Moreover, you can use a comment to mark a piece of code that is not needed at the moment (see the last line of the snippet below), e.g.:

# This program prints

# an introduction to the screen.

print("Hello!") # Invoking the print() function

# print("I'm Python.")

# Input and string

1. The print() function **sends data to the console**, while the input() function **gets data from the console**.

2. The input() function comes with an optional parameter: **the prompt string**. It allows you to write a message before the user input, e.g.:

name = input("Enter your name: ")

print("Hello, " + name + ". Nice to meet you!")

3. When the input() function is called, the program's flow is stopped, the prompt symbol keeps blinking (it prompts the user to take action when the console is switched to input mode) until the user has entered an input and/or pressed the *Enter* key.

name = input("Enter your name: ")

print("Hello, " + name + ". Nice to meet you!")

print("\nPress Enter to end the program.")

input()

print("THE END.")

4. The result of the input() function is a string. You can add strings to each other using the concatenation (+) operator. Check out this code:

num\_1 = input("Enter the first number: ") # Enter 12

num\_2 = input("Enter the second number: ") # Enter 21

print(num\_1 + num\_2) # the program returns 1221

5. You can also multiply (\* ‒ replication) strings, e.g.:

my\_input = input("Enter something: ") # Example input: hello

print(my\_input \* 3) # Expected output: hellohellohello

6. You can change type of input from string to number with:

num\_1 = int(input("Enter the first number: ") )# Enter 12

num\_2 = float(input("Enter the second number: ") )# Enter 21

#To convert number to string:

str(num\_1)

# Comparison Operators

The comparison (otherwise known as relational) operators are used to compare values. The table below illustrates how the comparison operators work, assuming that x = 0, y = 1, and z = 0:

|  |
| --- |
|  |
| **Operator** | **Description** | **Example** |
| == | returns True if operands' values are equal, and False otherwise |  |
| != | returns True if operands' values are not equal, and False otherwise | x != y # True  x != z # False |
| > | True if the left operand's value is greater than the right operand's value, and False otherwise | x > y # False  y > z # True |
| < | True if the left operand's value is less than the right operand's value, and False otherwise | x < y # True  y < z # False |
| ≥ | True if the left operand's value is greater than or equal to the right operand's value, and False otherwise | x >= y # False  x >= z # True  y >= z # True |
| ≤ | True if the left operand's value is less than or equal to the right operand's value, and False otherwise | x <= y # True  x <= z # True  y <= z # False |

# Conditional(if)

When you want to execute some code only if a certain condition is met, you can use a conditional statement:

a single if statement, e.g.:

x = 10

if x == 10: # condition

print("x is equal to 10") # Executed if the condition is True.

A series of if statements, e.g.:

x = 10

if x > 5: # condition one

print("x is greater than 5") # Executed if condition one is True.

if x < 10: # condition two

print("x is less than 10") # Executed if condition two is True.

if x == 10: # condition three

print("x is equal to 10") # Executed if condition three is True.

Each if statement is tested separately.

an if-else statement, e.g.:

x = 10

if x < 10: # Condition

print("x is less than 10") # Executed if the condition is True.

else:

print("x is greater than or equal to 10") # Executed if the condition is False.

a series of if statements followed by an else, e.g.:

x = 10

if x > 5: # True

print("x > 5")

if x > 8: # True

print("x > 8")

if x > 10: # False

print("x > 10")

else:

print("else will be executed")

Each if is tested separately. The body of else is executed if the last if is False.

The if-elif-else statement, e.g.:

x = 10

if x == 10: # True

print("x == 10")

if x > 15: # False

print("x > 15")

elif x > 10: # False

print("x > 10")

elif x > 5: # True

print("x > 5")

else:

print("else will not be executed")

If the condition for if is False, the program checks the conditions of the subsequent elif blocks – the first elif block that is True is executed. If all the conditions are False, the else block will be executed.

Nested conditional statements, e.g.:

x = 10

if x > 5: # True

if x == 6: # False

print("nested: x == 6")

elif x == 10: # True

print("nested: x == 10")

else:

print("nested: else")

else:

print("else")

# Loops

There are two types of loops in Python: while and for:

## While

The while loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

# Example 1

while True:

print("Stuck in an infinite loop.")

# Example 2

counter = 5

while counter > 2:

print(counter)

counter -= 1

## For

The for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function. Look at the examples below:

# Example 1

word = "Python"

for letter in word:

print(letter, end="\*")

# Example 2

for i in range(1, 10):

if i % 2 == 0:

print(i)

You can use the break and continue statements to change the flow of a loop:

* You use break to exit a loop, e.g.:

text = "OpenEDG Python Institute"

for letter in text:

if letter == "P":

break

print(letter, end="")

You use continue to skip the current iteration, and continue with the next iteration, e.g.:

text = "pyxpyxpyx"

for letter in text:

if letter == "x":

continue

print(letter, end="")

The while and for loops can also have an else clause in Python. The else clause executes after the loop finishes its execution as long as it has not been terminated by break, e.g.:

n = 0

while n != 3:

print(n)

n += 1

else:

print(n, "else")

print()

for i in range(0, 3):

print(i)

else:

print(i, "else")

The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

* start is an optional parameter specifying the starting number of the sequence (0 by default)
* stop is an optional parameter specifying the end of the sequence generated (it is not included),
* and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Example code:

for i in range(3):

print(i, end=" ") # Outputs: 0 1 2

for i in range(6, 1, -2):

print(i, end=" ") # Outputs: 6, 4, 2

# Logical and BitWise operators

Python supports the following logical operators:

* and → if both operands are true, the condition is true, e.g., (True and True) is True,
* or → if any of the operands are true, the condition is true, e.g., (True or False) is True,
* not → returns false if the result is true, and returns true if the result is false, e.g., not True is False.

2. You can use bitwise operators to manipulate single bits of data. The following sample data:

* x = 15, which is 0000 1111 in binary,
* y = 16, which is 0001 0000 in binary.

will be used to illustrate the meaning of bitwise operators in Python. Analyze the examples below:

* & does a *bitwise and*, e.g., x & y = 0, which is 0000 0000 in binary,
* | does a *bitwise or*, e.g., x | y = 31, which is 0001 1111 in binary,
* ˜  does a *bitwise not*, e.g., ˜ x = 240\*, which is 1111 0000 in binary,
* ^ does a *bitwise xor*, e.g., x ^ y = 31, which is 0001 1111 in binary,
* >> does a *bitwise right shift*, e.g., y >> 1 = 8, which is 0000 1000 in binary,
* << does a *bitwise left shift*, e.g., y << 3 = , which is 1000 0000 in binary,

\* -16 (decimal from signed 2's complement) -- read more about the [Two's complement](https://en.wikipedia.org/wiki/Two%27s_complement) operation.

# Lists

1. The **list is a type of data** in Python used to **store multiple objects**. It is an **ordered and mutable collection** of comma-separated items between square brackets, e.g.:

my\_list = [1, None, True, "I am a string", 256, 0]

2. Lists can be **indexed and updated**, e.g.:

my\_list = [1, None, True, 'I am a string', 256, 0]

print(my\_list[3]) # outputs: I am a string

print(my\_list[-1]) # outputs: 0

my\_list[1] = '?'

print(my\_list) # outputs: [1, '?', True, 'I am a string', 256, 0]

my\_list.insert(0, "first")

my\_list.append("last")

print(my\_list) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']

3. Lists can be **nested**, e.g.:

my\_list = [1, 'a', ["list", 64, [0, 1], False]]

You will learn more about nesting in module 3.1.7 - for the time being, we just want you to be aware that something like this is possible, too.

4. List elements and lists can be **deleted**, e.g.:

my\_list = [1, 2, 3, 4]

del my\_list[2]

print(my\_list) # outputs: [1, 2, 4]

del my\_list # deletes the whole list

5. Lists can be **iterated** through using the for loop, e.g.:

my\_list = ["white", "purple", "blue", "yellow", "green"]

for color in my\_list:

print(color)

6. The len() function may be used to **check the list's length**, e.g.:

my\_list = ["white", "purple", "blue", "yellow", "green"]

print(len(my\_list)) # outputs 5

del my\_list[2]

print(len(my\_list)) # outputs 4

7. A typical **function** invocation looks as follows: result = function(arg), while a typical **method** invocation looks like this:result = data.method(arg).

8. You can use the sort() method to sort elements of a list, e.g.:

lst = [5, 3, 1, 2, 4]

print(lst)

lst.sort()

print(lst) # outputs: [1, 2, 3, 4, 5]

9. There is also a list method called reverse(), which you can use to reverse the list, e.g.:

lst = [5, 3, 1, 2, 4]

print(lst)

lst.reverse()

print(lst) # outputs: [4, 2, 1, 3, 5]

10. If you have a list l1, then the following assignment: l2 = l1 does not make a copy of the l1 list, but makes the variables l1 and l2 **point to one and the same list in memory**. For example:

vehicles\_one = ['car', 'bicycle', 'motor']

print(vehicles\_one) # outputs: ['car', 'bicycle', 'motor']

vehicles\_two = vehicles\_one

del vehicles\_one[0] # deletes 'car'

print(vehicles\_two) # outputs: ['bicycle', 'motor']

11. If you want to copy a list or part of the list, you can do it by performing **slicing**:

colors = ['red', 'green', 'orange']

copy\_whole\_colors = colors[:] # copy the entire list

copy\_part\_colors = colors[0:2] # copy part of the list

12. You can use **negative indices** to perform slices, too. For example:

sample\_list = ["A", "B", "C", "D", "E"]

new\_list = sample\_list[2:-1]

print(new\_list) # outputs: ['C', 'D']

13. The start and end parameters are **optional** when performing a slice: list[start:end], e.g.:

my\_list = [1, 2, 3, 4, 5]

slice\_one = my\_list[2: ]

slice\_two = my\_list[ :2]

slice\_three = my\_list[-2: ]

print(slice\_one) # outputs: [3, 4, 5]

print(slice\_two) # outputs: [1, 2]

print(slice\_three) # outputs: [4, 5]

14. You can **delete slices** using the del instruction:

my\_list = [1, 2, 3, 4, 5]

del my\_list[0:2]

print(my\_list) # outputs: [3, 4, 5]

del my\_list[:]

print(my\_list) # deletes the list content, outputs: []

15. You can test if some items **exist in a list or not** using the keywords in and not in, e.g.:

my\_list = ["A", "B", 1, 2]

print("A" in my\_list) # outputs: True

print("C" not in my\_list) # outputs: True

print(2 not in my\_list) # outputs: False

## Multidimensional List

1. **List comprehension** allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

[expression for element in list if conditional]

which is actually an equivalent of the following code:

for element in list:

if conditional:

expression

Here's an example of a list comprehension ‒ the code creates a five-element list filled with the first five natural numbers raised to the power of 3:

cubed = [num \*\* 3 for num in range(5)]

print(cubed) # outputs: [0, 1, 8, 27, 64]

2. You can use **nested lists** in Python to create **matrices** (i.e., two-dimensional lists). For example:

# A four-column/four-row table ‒ a two dimensional array (4x4)

table = [[":(", ":)", ":(", ":)"],

[":)", ":(", ":)", ":)"],

[":(", ":)", ":)", ":("],

[":)", ":)", ":)", ":("]]

print(table)

print(table[0][0]) # outputs: ':('

print(table[0][3]) # outputs: ':)'

3. You can nest as many lists in lists as you want, thereby creating n-dimensional lists, e.g., three-, four- or even sixty-four-dimensional arrays. For example:

# Cube - a three-dimensional array (3x3x3)

cube = [[[':(', 'x', 'x'],

[':)', 'x', 'x'],

[':(', 'x', 'x']],

[[':)', 'x', 'x'],

[':(', 'x', 'x'],

[':)', 'x', 'x']],

[[':(', 'x', 'x'],

[':)', 'x', 'x'],

[':)', 'x', 'x']]]

print(cube)

print(cube[0][0][0]) # outputs: ':('

print(cube[2][2][0]) # outputs: ':)'

# Functions

1. A function is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

2. There are at least four basic types of functions in Python:

**built-in functions** which are an integral part of Python (such as the print() function). You can see a complete list of Python built-in functions at <https://docs.python.org/3/library/functions.html>.

The ones that come **from pre-installed modules** (you'll learn about them in the Python Essentials 2 course)

**user-defined functions** which are written by users for users - you can write your own functions and use them freely in your code,

**the lambda functions**

3. You can define your own function using the def keyword and the following syntax:

def your\_function(optional parameters):

# the body of the function

You can define a function which doesn't take any arguments, e.g.:

def message(): # defining a function

print("Hello") # body of the function

message() # calling the function

4. You can pass information to functions by using parameters. Your functions can have as many parameters as you need.

An example of a one-parameter function:

def hi(name):

print("Hi,", name)

hi("Greg")

An example of a three-parameter function:

def address(street, city, postal\_code):

print("Your address is:", street, "St.,", city, postal\_code)

s = input("Street: ")

p\_c = input("Postal Code: ")

c = input("City: ")

address(s, c, p\_c)

5. You can pass arguments to a function using the following techniques:

**positional argument passing** in which the order of arguments passed matters

Ex. 1

def subtra(a, b):

print(a - b)

subtra(5, 2) # outputs: 3

subtra(2, 5) # outputs: -3

**keyword (named) argument passing** in which the order of arguments passed doesn't matter (Ex. 2),

a mix of positional and keyword argument passing (Ex. 3).

Ex. 2

def subtra(a, b):

print(a - b)

subtra(a=5, b=2) # outputs: 3

subtra(b=2, a=5) # outputs: 3

Ex. 3

def subtra(a, b):

print(a - b)

subtra(5, b=2) # outputs: 3

subtra(5, 2) # outputs: 3

It's important to remember that positional arguments mustn't follow keyword arguments. That's why if you try to run the following snippet:

def subtra(a, b):

print(a - b)

subtra(5, b=2) # outputs: 3

subtra(a=5, 2) # Syntax Error

6. You can use the keyword argument passing technique to pre-define a value for a given argument:

def name(first\_name, last\_name="Smith"):

print(first\_name, last\_name)

name("Andy") # outputs: Andy Smith

name("Betty", "Johnson") # outputs: Betty Johnson (the keyword argument replaced by "Johnson")