Python Programming Fundamentals

DA2303 – Python Programming

Table of contents

01

Conditions, Branching & Loops

if statement, while statement, for loop, break statement and continue statement

Conditions, Branching & Loops

- In the programs we have seen till now, there has always been a series of statements faithfully executed by Python in exact top-down order.
- What if you wanted to change the flow of how it works?
- For example, you want the program to take some decisions and do different things depending on different situations, such as printing 'Good Morning' or 'Good Evening' depending on the time of the day?
- As you might have guessed, this is achieved using control flow statements.
- There are three control flow statements in Python - if, for and while.

The if statement

- The if statement is used to check a condition: *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*).

- The *else* clause is optional.

Example: if statement

- save as if.py

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here
print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

Example: if statement (cont.)

- Run the script 3 times and enter any int value.

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done

$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done

$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

Example: if statement – How it works

- In this program, we take guesses from the user and check if it is the number that we have.
- We set the variable number to any integer we want, say 23.
- Then, we take the user's guess using the input() function. Functions are just reusable pieces of programs.
- We supply a string to the built-in input function which prints it to the screen and waits for input from the user.

Example: if statement – How it works (cont.)

- Once we enter something and press [enter] key, the input() function returns what we entered, as a string.
- We then convert this string to an integer using int and then store it in the variable guess.
- Actually, the int is a class but all you need to know right now is that you can use it to convert a string to an integer (assuming the string contains a valid integer in the text).

Example: if statement – How it works (cont.)

- Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message.
- Notice that we use indentation levels to tell Python which statements belong to which block.
- This is why indentation is so important in Python. I hope you are sticking to the "consistent indentation" rule.
- Notice how the if statement contains a colon at the end - we are indicating to Python that a block of statements follows.

Example: if statement – How it works (cont.)

- Then, we check if the guess is less than the number, and if so, we inform the user that they must guess a little higher than that.
- What we have used here is the elif clause which actually combines two related if else-if else statements into one combined if-elif-else statement.
- This makes the program easier and reduces the amount of indentation required.
- The elif and else statements must also have a colon at the end of the logical line followed by their corresponding block of statements (with proper indentation, of course)

Example: if statement – How it works (cont.)

- You can have another if statement inside the if-block of an if statement and so on - this is called a nested if statement.

- Remember that the elif and else parts are optional. A minimal valid if statement is:

```
if True:
    print('Yes, it is true')
```

Example: if statement – How it works (cont.)

- After Python has finished executing the complete if statement along with the associated elif and else clauses, it moves on to the next statement in the block containing the if statement.

- In this case, it is the main block (where execution of the program starts), and the next statement is the print('Done') statement.

- After this, Python sees the ends of the program and simply finishes up.

Example: if statement – How it works (cont.)

- Even though this is a very simple program, I have been pointing out a lot of things that you should notice.
- All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds).
- You will need to become aware of all these things initially, but after some practice you will become comfortable with them, and it will all feel 'natural' to you.

The while statement

- The while statement allows you to repeatedly execute a block of statements as long as a condition is true.
- A while statement is an example of what is called a *looping* statement.
- A while statement can have an optional else clause.

Example: while statement

- save as while.py

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')

else:
    print('The while loop is over.')
    # Do anything else you want to do here
print('Done')
```

Example: while statement (cont.)

- Run the script.

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

Example: while statement – How it works

- In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly run the program for each guess, as we have done in the previous section.

- This aptly demonstrates the use of the while statement.

- We move the input and if statements to inside the while loop and set the variable running to True before the while loop.

Example: while statement – How it works (cont.)

- First, we check if the variable running is True and then proceed to execute the corresponding *while-block*.
- After this block is executed, the condition is again checked which in this case is the running variable.
- If it is true, we execute the while-block again, else we continue to execute the optional else-block and then continue to the next statement.
- The else block is executed when the while loop condition becomes False - this may even be the first time that the condition is checked.

Example: while statement – How it works (cont.)

- If there is an else clause for a while loop, it is always executed unless you break out of the loop with a break statement.

- The True and False are called Boolean types and you can consider them to be equivalent to the value 1 and 0 respectively.

The for loop

- The for..in statement is another looping statement which *iterates* over a sequence of objects i.e. go through each item in a sequence.
- We covered sequence already.
- Review: A sequence is an ordered collection of items.

Example: for loop

- Save as for.py

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

- Run the script

```
$ python for.py
1
2
3
4
The for loop is over
```

Example: for loop – How it works

- In this program, we are printing a *sequence* of numbers.
- We generate this sequence of numbers using the built-in range function.
- What we do here is supply it two numbers and range returns a sequence of numbers starting from the first number and up to the second number.
- For example, range(1,5) gives the sequence [1, 2, 3, 4].
- By default, range takes a step count of 1.

Example: for loop – How it works (cont.)

- If we supply a third number to range, then that becomes the step count. For example, range(1,5,2) gives [1,3].

- Remember that the range extends *up to* the second number i.e. it does *not* include the second number.

- Note that range() generates only one number at a time, if you want the full list of numbers, call list() on the range(), for example, list(range(5)) will result in [0, 1, 2, 3, 4].

Example: for loop – How it works (cont.)

- The for loop then iterates over this range - for i in range(1,5) is equivalent to for i in [1, 2, 3, 4] which is like assigning each number (or object) in the sequence to i, one at a time, and then executing the block of statements for each value of i.

- In this case, we just print the value in the block of statements.

- Remember that the else part is optional. When included, it is always executed once after the for loop is over unless a break statement is encountered.

Example: for loop – How it works (cont.)

- Remember that the for..in loop works for any sequence.
- Here, we have a list of numbers generated by the built-in range function, but in general we can use any kind of sequence of any kind of objects!

The break statement

- The break statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has not been completely iterated over.

- An important note is that if you *break* out of a for or while loop, any corresponding loop else block is **not** executed.

Example: break statement

- save as break.py

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

Example: break statement (cont.)

- Run the script and enter what you want.

```
$ python break.py
Enter something : Programming is fun
Length of the string is 18
Enter something : When the work is done
Length of the string is 21
Enter something : if you wanna make your work also fun:
Length of the string is 37
Enter something : use Python!
Length of the string is 11
Enter something : quit
Done
```

Example: break statement – How it works

- In this program, we repeatedly take the user's input and print the length of each input each time.
- We are providing a special condition to stop the program by checking if the user input is 'quit'.
- We stop the program by *breaking* out of the loop and reach the end of the program.
- The length of the input string can be found out using the built-in len function.
- Remember that the break statement can be used with the for loop as well.

The continue statement

- The continue statement is used to tell Python to skip the rest of the statements in the current loop block and to *continue* to the next iteration of the loop.

Example: continue statement

- save as continue.py

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
    print('Input is of sufficient length')
    # Do other kinds of processing here...
```

Example: continue statement (cont.)

- Run the script.

```
$ python continue.py
Enter something : a
Too small
Enter something : 12
Too small
Enter something : abc
Input is of sufficient length
Enter something : quit
```

Example: continue statement – How it works

- In this program, we accept input from the user, but we process the input string only if it is at least 3 characters long.
- So, we use the built-in len function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the continue statement.
- Otherwise, the rest of the statements in the loop are executed, doing any kind of processing we want to do here.
- Note that the continue statement works with the for loop as well.

02

Functions

Functions, Parameters,
Local Variables

Functions

- Functions are reusable pieces of programs.
- They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in your program and any number of times.
- This is known as *calling* the function.
- We have already used many built-in functions such as len and range.
- The function concept is probably *the* most important building block of any non-trivial software (in any programming language), so we will explore various aspects of functions in this chapter.

Functions (cont.)

- Functions are defined using the def keyword.
- After this keyword comes an *identifier* name for the function, followed by a pair of parentheses which may enclose some names of variables, and by the final colon that ends the line.
- Next follows the block of statements that are part of this function.

Example: Functions

- save as function1.py

```
def say_hello():
    # block belonging to the function
    print('hello world')
# End of function

say_hello() # call the function
say_hello() # call the function again
```

- Run the script.

```
$ python function1.py
hello world
hello world
```

Example: Functions – How it works

- We define a function called say_hello using the syntax as explained above.
- This function takes no parameters and hence there are no variables declared in the parentheses.
- Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.
- Notice that we can call the same function twice which means we do not have to write the same code again.

Function Parameters

- A function can take parameters, which are values you supply to the function so that the function can *do* something utilising those values.
- These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs.
- Parameters are specified within the pair of parentheses in the function definition, separated by commas.
- When we call the function, we supply the values in the same way.
- Note the terminology used - the names given in the function definition are called *parameters* whereas the values you supply in the function call are called *arguments*.

Example: Function Parameters

- save as function_param.py

```
def print_max(a, b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')
# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

Example: Function Parameters (cont.)

- Run the script.

```
$ python function_param.py
4 is maximum
7 is maximum
```

Example: Function Parameters – How it works

- Here, we define a function called print_max that uses two parameters called a and b. We find out the greater number using a simple if..else statement and then print the bigger number.
- The first time we call the function print_max, we directly supply the numbers as arguments.
- In the second case, we call the function with variables as arguments.
- print_max(x, y) causes the value of argument x to be assigned to parameter a and the value of argument y to be assigned to parameter b.
- The print_max function works the same way in both cases.

Local Variables

- When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are *local* to the function.
- This is called the *scope* of the variable.
- All variables have the scope of the block they are declared in starting from the point of definition of the name.

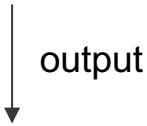
Example: Local Variables

- save as function_local.py

```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```



- Run the script

```
$ python function_local.py
x is 50
Changed local x to 2
x is still 50
```

Example: Local Variables – How it works

- The first time that we print the *value* of the name *x* with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.
- Next, we assign the value 2 to x.
- The name x is local to our function.
- So, when we change the value of x in the function, the x defined in the main block remains unaffected.
- With the last print statement, we display the value of x as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The global statement

- If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is *global*.
- We do this using the global statement.
- It is impossible to assign a value to a variable defined outside a function without the global statement.

The global statement (cont.)

- You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function).
- However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is.
- Using the global statement makes it amply clear that the variable is defined in an outermost block.

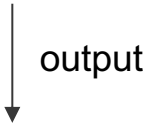
Example: global statement

- save as function_global.py

```
x = 50

def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)

func()
print('Value of x is', x)
```



- Run the script.

```
$ python function_global.py
x is 50
Changed global x to 2
Value of x is 2
```

Example: global statement – How it works

- The global statement is used to declare that x is a global variable - hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block.

- You can specify more than one global variable using the same global statement e.g. global x, y, z.

Default Argument Values

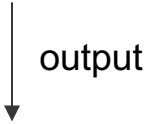
- For some functions, you may want to make some parameters *optional* and use default values in case the user does not want to provide values for them.
- This is done with the help of default argument values.
- You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value.
- Note that the default argument value should be a constant.
- More precisely, the default argument value should be immutable.

Example: Default Argument Values

- save as function_default.py

```
def say(message, times=1):
    print(message * times)

say('Hello')
say('World', 5)
```



- Run the script.

```
$ python function_default.py
Hello
WorldWorldWorldWorldWorld
```

Example: Default Argument Values – How it works

- The function named say is used to print a string as many times as specified.
- If we don't supply a value, then by default, the string is printed just once.
- We achieve this by specifying a default argument value of 1 to the parameter times.
- In the first usage of say, we supply only the string and it prints the string once.
- In the second usage of say, we supply both the string and an argument 5 stating that we want to *say* the string message 5 times.

Example: Default Argument Values – CAUTION!

- Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list.

- This is because the values are assigned to the parameters by position.

- For example,
  - def func(a, b=5) is valid
  - def func(a=5, b) is *not valid*.

Keyword Arguments

- If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called *keyword arguments* - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

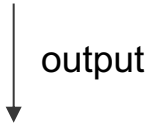
- There are two advantages –
  - One, using the function is easier since we do not need to worry about the order of the arguments.
  - Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

Example: Keyword Arguments

- save as function_keyword.py

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```



- Run the script.

```
$ python function_keyword.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Example: Keyword Arguments – How it works

- The function named func has one parameter without a default argument value, followed by two parameters with default argument values.

- In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 7 and c gets the default value of 10.

- In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument.
  - Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments.
  - The variable b gets the default value of 5.

Example: Keyword Arguments – How it works (cont.)

- In the third usage func(c=50, a=100), we use keyword arguments for all specified values.
  - Notice that we are specifying the value for parameter c before that for a even though a is defined before c in the function definition.

VarArgs parameters

- Sometimes you might want to define a function that can take *any* number of parameters, i.e. **var**iable number of **arg**uments, this can be achieved by using the stars.

Example: VarArgs parameters

- Sometimes you might want to define a function that can take *any* number of parameters, i.e. **var**iable number of **arg**uments, this can be achieved by using the stars (save as function_varargs.py):

```
def total(a=5, *numbers, **phonebook):
    print('a', a)
    #iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item)
    #iterate through all the items in dictionary
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

total(10,1,2,3,Jack=1123,John=2231,Inge=1560)
```

Example: VarArgs parameters (cont.)

- Run the script

```
$ python function_varargs.py
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
```

Example: VarArgs parameters – How it works

- When we declare a starred parameter such as *param, then all the positional arguments from that point till the end are collected as a tuple called 'param'.

- Similarly, when we declare a double-starred parameter such as **param, then all the keyword arguments from that point till the end are collected as a dictionary called 'param'.

The return statement

- The return statement is used to *return* from a function i.e. break out of the function. We can optionally *return a value* from the function as well.

Example: return statement

- save as function_return.py

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

- Run the script

```
$ python function_return.py
3
```

Example: return statement – How it works

- The maximum function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple if..else statement to find the greater value and then *returns* that value.

- Note that a return statement without a value is equivalent to return None.

- None is a special type in Python that represents nothingness.

- For example, it is used to indicate that a variable has no value if it has a value of None.

Example: return statement – How it works (cont.)

- Every function implicitly contains a return None statement at the end unless you have written your own return statement.
- You can see this by running print(some_function()) where the function some_function does not use the return statement such as:

```
def some_function():
    pass
```

- The pass statement is used in Python to indicate an empty block of statements.

03

Objects and Classes

Objects, Classes, Methods, Class and Object Variables

Object Oriented Programming

- In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data.
- This is called the *procedure-oriented* way of programming.
- There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object.
- This is called the *object oriented* programming paradigm.
- Most of the time you can use procedural programming, but when writing large programs or have a problem that is better suited to this method, you can use object oriented programming techniques.

Object Oriented Programming (cont.)

- Classes and objects are the two main aspects of object oriented programming.
- A **class** creates a new *type* where **objects** are **instances** of the class.
- An analogy is that you can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class.
- Note that integers are treated as objects (of the int class) in Python.

Object Oriented Programming (cont.)

- Objects can store data using ordinary variables that *belong* to the object.
- Variables that belong to an object or class are referred to as **fields**.
- Objects can also have functionality by using functions that *belong* to a class.
- Such functions are called **methods** of the class.
- This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object.

Object Oriented Programming (cont.)

- Collectively, the fields and methods can be referred to as the **attributes** of that class.
- Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself.
- They are called **instance variables** and **class variables** respectively.
- A class is created using the class keyword. The fields and methods of the class are listed in an indented block.

The self

- Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it.

- This particular variable refers to the object *itself*, and by convention, it is given the name self.

- Although, you can give any name for this parameter, it is *strongly recommended* that you use the name self - any other name is definitely frowned upon.

- There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use self.

The self (cont.)

- You must be wondering how Python gives the value for self and why you don't need to give a value for it.

- An example will make this clear.
  - Say you have a class called MyClass and an instance of this class called myobject.
  - When you call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) - this is all the special self is about.

- This also means that if you have a method which takes no arguments, then you still have to have one argument - the self.

Classes

- The simplest class possible is shown in the following example (save as oop_simplestclass.py).

```
class Person:
    pass # An empty block

p = Person()
print(p)
```

- Run the script

```
$ python oop_simplestclass.py
<__main__.Person instance at 0x10171f518>
```

Classes – How it works

- We create a new class using the class statement and the name of the class.
- This is followed by an indented block of statements which form the body of the class.
- In this case, we have an empty block which is indicated using the pass statement.
- Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses.

Classes – How it works

- For our verification, we confirm the type of the variable by simply printing it.
- It tells us that we have an instance of the Person class in the __main__ module.
- Notice that the address of the computer memory where your object is stored is also printed.
- The address will have a different value on your computer since Python can store the object wherever it finds space.

Methods

- We have already discussed that classes/objects can have methods just like functions except that we have an extra self variable. We will now see an example (save as oop_method.py).

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

- Run the script.

output

```
$ python oop_method.py
Hello, how are you?
```

Methods – How it works

- Here we see the self in action.
- Notice that the say_hi method takes no parameters but still has the self in the function definition.

The __init__ method

- There are many method names which have special significance in Python classes.
- We will see the significance of the __init__ method now.
- The __init__ method is run as soon as an object of a class is instantiated (i.e. created).
- The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object.
- Notice the double underscores both at the beginning and at the end of the name.

Example: __init__ method

- save as oop_init.py

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
# The previous 2 lines can also be written as
# Person('Swaroop').say_hi()
```

- Run the script.

```
$ python oop_init.py
Hello, my name is Swaroop
```

Example: __init__ method – How it works

- Here, we define the __init__ method as taking a parameter name (along with the usual self).
- Here, we just create a new field also called name.
- Notice these are two different variables even though they are both called 'name'.
- There is no problem because the dotted notation self.name means that there is something called "name" that is part of the object called "self" and the other name is a local variable.
- Since we explicitly indicate which name we are referring to, there is no confusion.

Example: __init__ method – How it works (cont.)

- When creating new instance p, of the class Person, we do so by using the class name, followed by the arguments in the parentheses: p = Person('Swaroop').
- We do not explicitly call the __init__ method.
- This is the special significance of this method.
- Now, we are able to use the self.name field in our methods which is demonstrated in the say_hi method.

Class And Object Variables

- We have already discussed the functionality part of classes and objects (i.e. methods), now let us learn about the data part.
- The data part, i.e. fields, are nothing but ordinary variables that are *bound* to the **namespaces** of the classes and objects.
- This means that these names are valid within the context of these classes and objects only.
- That's why they are called *name spaces*.

Class And Object Variables (cont.)

- There are two types of *fields* - class variables and object variables which are classified depending on whether the class or the object *owns* the variables respectively.

- **Class variables** are shared - they can be accessed by all instances of that class.
  - There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.

- **Object variables** are owned by each individual object/instance of the class.
  - In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance.

Example: Class And Object Variables

- save as oop_objvar.py

```
class Robot:
    """Represents a robot, with a name."""

    # A class variable, counting the number of robots
    population = 0

    def __init__(self, name):
        """Initializes the data."""
        self.name = name
        print("(Initializing {})".format(self.name))

        # When this person is created, the robot
        # adds to the population
        Robot.population += 1
```

- Continue next slide.

```
    def die(self):
        """I am dying."""
        print("{} is being destroyed!".format(self.name))

        Robot.population -= 1

        if Robot.population == 0:
            print("{} was the last one.".format(self.name))
        else:
            print("There are still {:d} robots working.".format( Robot.population))

    def say_hi(self):
        """Greeting by the robot.

        Yeah, they can do that."""
        print("Greetings, my masters call me {}.".format(self.name))
```

Continue next slide.

```
    @classmethod
    def how_many(cls):
        """Prints the current population."""
        print("We have {:d} robots.".format(cls.population))

droid1 = Robot("R2-D2")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()

print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")
droid1.die()
droid2.die()

Robot.how_many()
```

Example: Class And Object Variables (cont.)

- Run the script.

```
$ python oop_objvar.py
(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

Example: Class And Object Variables – How it works

- This is a long example but helps demonstrate the nature of class and object variables.
- Here, population belongs to the Robot class and hence is a class variable.
- The name variable belongs to the object (it is assigned using self) and hence is an object variable.
- Thus, we refer to the population class variable as Robot.population and not as self.population.
- We refer to the object variable name using self.name notation in the methods of that object.

Example: Class And Object Variables – How it works (cont.)

- Remember this simple difference between class and object variables.
- Also note that an object variable with the same name as a class variable will hide the class variable!
- Instead of Robot.population, we could have also used self.__class__.population because every object refers to its class via the self.__class__ attribute.

Example: Class And Object Variables – How it works (cont.)

- The how_many is actually a method that belongs to the class and not to the object.
- This means we can define it as either a classmethod or a staticmethod depending on whether we need to know which class we are part of.
- Since we refer to a class variable, let's use classmethod.
- We have marked the how_many method as a class method using a decorator.

Example: Class And Object Variables – How it works (cont.)

- Decorators can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the @classmethod decorator is the same as calling:

```
how_many = classmethod(how_many)
```

- Observe that the __init__ method is used to initialize the Robot instance with a name. In this method, we increase the population count by 1 since we have one more robot being added.

- Also observe that the values of self.name is specific to each object which indicates the nature of object variables.

Example: Class And Object Variables – How it works (cont.)

- Remember, that you must refer to the variables and methods of the same object using the self *only*.
- This is called an *attribute reference*.
- In the die method, we simply decrease the Robot.population count by 1.
- All class members are public.
- One exception: If you use data members with names using the *double underscore prefix* such as __privatevar, Python uses name-mangling to effectively make it a private variable.

Example: Class And Object Variables – How it works (cont.)

- Thus, the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects.

- Remember that this is only a convention and is not enforced by Python (except for the double underscore prefix).

Thanks

Do you have any question?

jailani.rahman@pb.edu.bn
(+673) 223 4466 ext 241
Unit 6.01, Ong Sum Ping Campus,
Politeknik Brunei.