01 - Python_Fundamentals_Course_Notes

November 22, 2021

1 Python Fundamentals

Instructor: Himalaya Kakshapati

Welcome to the Python Fundamentals course.

2 Python Hello World

When learning any new programming language, it is customary to write a Hello World! statement and run it. Keeping with the tradition, let's write our Hello World! statement in Python and run it.

2.0.1 Print "Hello World"

In order to print "Hello World!", type the syntax: `print("Hello World!")` on Colab notebook and press Ctrl+Enter

[ ]:
```
print("Hello World!")
```

2.0.2 Exercise 1.1:

Write the syntax to print "I love Python" and run it.

[ ]:
```
# write your code to print "I love Python" below and run it.
```

2.1 Python comments

Comments are descriptions used to briefly describe the code. It is always a good idea to include comments in your code. Code are instructions for the computer, while comments are information for humans. Your code should be understandable to other people, so that other people can maintain and extend your code.

Single line Comments If you want single line comment, you need to put a "#" symbol before the comment. Example is shown below.

[ ]:
```
# This is a single line comment. Always comment your code.
```

Multi line comments If your comment is going to be more than one line, you can enclose them with """ as shown in the example below.

[ ]:
```
""" This is multi-line comment
This is another line of the comment.
Here is another one. """
```

3 Python Variables

Variables are containers that hold data values. Please look at the examples of variables below.

[ ]:
```
# examples of variables
x = 2 # the variable x holds the value 2

# variables are case-sensitive (i.e. variables x and X are different)
X = 5 # this variable X is different from x above

# You will see that the values of x and X are different
print('x =', x)
print('X =', X)
```

[ ]:
```
# variables can hold string values (text) as well. String needs to be enclosed␣
↪inside a pair of single or double quotes.
language = 'Python'  # string inside single quotes
name = "John" # string inside double quotes

# when you print variables, the values that they hold gets printed
print('language: ', language)
print('name:', name)
```

```
language:  Python
name: John
```

Rules for variable names

- Variable name should start with either a letter or an underscore. Example: `age` and `_age` are valid variable names.
- Variable name can not start with a number. For example, `7name` is not valid. However, number can be included after the first character. For example, `name7` or `n7ame` are valid.
- Variable name can only contain alpha-numeric ([a-z], [A-Z], [0,9]) or underscore (_) characters
- Variable names are case-sensitive. Example: `address`, `Address`, `aDdress` are different variables.

[ ]:
```
# the code below will show an error when you run it
7name = "Harry" # variable name cannot start with a number, so you will get an␣
↪error.
```

[ ]:
```
# the following variable names are valid
_age = 25
name7 = "Sarah" # number can be anywhere after the first character
```

```
n7ame = 'Johnson' # number can be anywhere after the first character

# variable names are case-sensitive. So the following two variables are␣
 ↪different although they have the same spelling.
country = "Nepal"
Country = "USA"

# print the variables
print("_age:", _age)
print("name7:", name7)
print("n7ame:", n7ame)
print("country:", country)
print("Country:", Country)
```

Exercise 2.1 Create variables named 'name' and 'age' and assign your name and age respectively. Then print the variables. Also, put appropriate comments in your code.

[ ]:
```
# create a variable 'name' and assign your name to it

# create a variable 'age' and assign your age to it

# print the above two variables
```

3.1 Python Data Types

Variables can hold different types of data.

Python has the following data types. * Text (String): `str` * Numeric (Numbers): `int`, `float`, `complex` * Sequence: `list`, `tuple`, `range` * Mapping: `dict` * Set: `set`, `frozenset` * Boolean (True/False): `bool` * Binary (ones/zeros): `bytes`, `bytearray`, `memoryview`

Examples of commonly used data types

[ ]:
```
name = "Sam" # str type (string)
age = 25 # int type (integer)
height = 2.45 # float type (floating point)
fruits = ["Apple", "Banana", "Pear"] # list type (use square brackets)
countries = {"China", "India", "Nepal"} # set type (use curly braces)
colors = ("red", "green", "blue") # tuple type (use parentheses)
x = range(5) # range type
my_info = {name: "Sam", age: 25} # dict type (key:value pairs inside a pair of␣
 ↪curly braces)
is_ok = True # boolean type (either True or False)
```

You can use type() to show the type of the variable as shown below.

[ ]:
```
print("Type of name variable:",type(name))
print("Type of my_info variable:",type(my_info))
print("Type of is_ok variable:",type(is_ok))
```

```
Type of name variable: <class 'str'>
Type of my_info variable: <class 'dict'>
Type of is_ok variable: <class 'bool'>
```

##Python Type casting You can create a variable of particular type, or convert one type to another type, by something called casting. Here are some examples.

[ ]:
```
int_var = int(2.53) # int_var will contain the value int value 2 (numbers after␣
 ↪the decimal are discarded)
str_var = str(45) # str_var will contain the value "45" in string form
float_var = float("3.14") # float_var will contain the value 3.14 in floating␣
 ↪point format

# you can simply assign the values to the variables and the type is␣
 ↪automatically inferred
int_var_auto = 4
str_var_auto = "hello"
float_var_auto = 3.14

print("Type of int_var_auto", type(int_var_auto))
print("Type of str_var_auto", type(str_var_auto))
print("Type of float_var_auto", type(float_var_auto))
```

```
Type of int_var_auto <class 'int'>
Type of str_var_auto <class 'str'>
Type of float_var_auto <class 'float'>
```

[ ]:
```
# let's convert int_var to float type
int_var_convert_float = float(int_var)
print(int_var_convert_float)
print(type(int_var_convert_float))
```

```
2.0
<class 'float'>
```

3.2 Exercise 2.1

Create a variable of type `int` and convert it to `float` and `str` type.

[ ]:
```
# write code below to create a variable of type int

# write code below to convert the above int variable to float

# write code below to convert the int variable you created above to str type
```

4 Python Operators

Operators are symbols that help perform operations between values and variables. Python has the following groups of operators.

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

4.1 Arithmetic Operators

[1]:
```
# Addition (+)
2 + 3
```

[1]: 5

[2]:
```
# Subtraction (-)
4 - 6
```

[2]: -2

[ ]:
```
# Multiplication ( * symbol is used for multiplication)
4 * 7
```

[ ]: 28

[ ]:
```
# Division ( / symbol is used for division)
3 / 4
```

[ ]: 0.75

[ ]:
```
# Modulus ( % )
4 % 3 # ( % yields the remainder of the division); Here the result is 1, which␣
 ↪is the remainder when 4 is divided by 3.
```

[ ]: 1

[ ]:
```
# Exponentiation ( ** ) or power
2 ** 3 # the result is 8 since 2 to the power of 3 is 2x2x2 = 8
```

[ ]: 8

[ ]:
```
# Floor division ( // )
```

```
5 // 3 # the result is 1, since 5/3 = 1.66... and only the number before the␣
↪decimal is returned as the result
```

[ ]: 1

4.2 Exercise 3.1

Write the following expressions in Python code. * x = 7(3+8-4)/6 * y = int(x) modulus 2 * z = y to the power of 3 * k = z floor divided by 3

Replace 'None' with your code in the following exercises.

[ ]:
```
# write code below for x = 7(3+8-4)/6. Hint: 3(m + n) in Python code is 3*(m+n)␣
↪ RESULT should be: 8.166666666666666
x = None
x
```

[ ]:
```
# write code below for y = int(x) modulus 2 , RESULT should be: 2
y = None
y
```

[ ]:
```
# write code below for z = y to the power of 3, RESULT should be: 8
z = None
z
```

[ ]:
```
# write code below for z floor divided by 3, RESULT should be: 2
k = None
k
```

4.3 Assignment Operators

[ ]:
```
# = operator assigns the value on the right hand side to the variable on the␣
↪left hand side
x = 5
x
```

[ ]: 5

[ ]:
```
# +=
x += 3 # same as x = x + 3 # RESULT should be 8
x
```

[ ]: 8

[ ]:
```
# -=
x -= 3 # same as x = x - 3 # RESULT should be 5
x
```

[ ]: 5

[ ]:
```
# *=
x *= 3 # same as x = x * 3 # RESULT should be 15
x
```

[ ]: 15

[ ]:
```
# /=
x /= 3 # same as x = x / 3 # RESULT should be 5.0
x
```

[ ]: 5.0

[ ]:
```
# %=
x %= 3 # same as x = x % 3 # RESULT should be 2.0
x
```

[ ]: 2.0

[ ]:
```
# //=
x = 8
x //= 3 # same as x = x - 3 # RESULT should be 2
x
```

[ ]: 2

[ ]:
```
# **=
x **= 3 # same as x = x ** 3 # RESULT should be 8
x
```

[ ]: 8

4.4 Comparison Operators

Comparison operators produce a result as boolean (True/False) values.

[ ]:
```
# == (Equal to)
2 == 3 # False
```

[ ]: False

[ ]:
```
# != (Not equal to)
2 != 3 # True
```

[ ]: True

[ ]:
```
# > (Greater than)
2 > 3 # False
```

[ ]:
```
False
```

[ ]:
```
# < (Less than)
2 < 3 # True
```

[ ]:
```
True
```

[ ]:
```
# >= (Greater than or equal to)
2 >= 3 # False
```

[ ]:
```
False
```

[ ]:
```
# <= (Less than or equal to)
2 <= 2 # True
```

[ ]:
```
True
```

4.5 Logical Operators

In Python, Logical operators are used on conditional statements (either `True` or `False`).

They perform * Logical AND, * Logical OR, and * Logical NOT

operations.

- `and` => (Logical AND): `True` if both the operands are true => example: `x and y`
- `or` => (Logical OR): `True` if either of the operands is true => example: `x or y`
- `not` => (Logical NOT): `True` if operand is false => example: `not x`

[ ]:
```
a = 2
b = 3
```

[ ]:
```
a > 2 and b > 2 # False and True = False
```

[ ]:
```
False
```

[ ]:
```
a > 1 and b > 2 # True and True = True
```

[ ]:
```
True
```

[ ]:
```
a < 1 or b > 1 # False or True = True
```

[ ]:
```
True
```

[ ]:
```
a < 1 or b < 3 # False or False = False
```

[ ]: False

```
[ ]: not a > 1 # not True = False
```

[ ]: False

```
[ ]: not b < 3 # not False = True
```

[ ]: True

Short-circuit Operations

```
[2]: False and whatever
```

[2]: False

```
[1]: '' and whatever # '' (empty string) is a boolean False
```

[1]: ''

```
[3]: [] and whatever # empty list, tuple, set, dict etc are all boolean False
```

[3]: []

```
[4]: True or whatever
```

[4]: True

```
[5]: 'Hello' or whatever
```

[5]: 'Hello'

Logical Operators Precedence Rules

Logical Operators precedence rules are as follows.

1. not
2. and
3. or

```
[8]: False or True and not True
```

[8]: False

4.6 Identity Operators

The identity operators in Python are used to determine whether a value is of a certain class or type.

They are usually used to determine the type of data a certain variable contains. For example, you can combine the identity operators with the built-in type() function to ensure that you are working with the specific variable type.

Two identity operators are available in Python:

- `is` – returns `True` if the type of the value in the right operand points to the same type in the left operand.

For example, `type(3) is int` evaluates to `True` because `3` is indeed an integer number. * `is not` – returns `True` if the type of the value in the right operand points to a different type than the value in the left operand.

For example, `type(3) is not float` evaluates to `True` because `3` is not a floating-point value.

[ ]:
```
x = 3

type(x) is int # type of x is indeed int, so we get True
```

[ ]:
```
True
```

[ ]:
```
type(x) is float # type of x is not float, so we get False
```

[ ]:
```
False
```

[ ]:
```
type(x) is not str # type of x is not str, so we get True
```

[ ]:
```
True
```

4.7 Membership Operators

Membership operators are used to test if a sequence is present in an object.

The two membership operators in Python are: * `in` * `not in`

[ ]:
```
'h' in 'hello' # 'h' is in 'hello', so we get True
```

[ ]:
```
True
```

[ ]:
```
'ell' in 'hello' # 'ello' is in 'hello', so we get True
```

[ ]:
```
True
```

[ ]:
```
'm' in 'python' # 'm' is not in 'python', so we get False
```

[ ]:
```
False
```

4.8 Bitwise Operators

In Python, bitwise operators are used to performing bitwise calculations on integers. The integers are first converted into binary and then operations are performed on bit by bit, hence the name bitwise operators. Then the result is returned in decimal format.

Note: Python bitwise operators work only on integers.

[ ]:
```
# Bitwise and (&)
2 & 3
# 2 = 10 in binary
# 3 = 11 in binary
# 2 & 3 = 10 & 11 = 10 => 2 in decimal
```

[ ]:
```
2
```

[ ]:
```
# Bitwise or (|)
2 | 3
# 2 = 10 in binary
# 3 = 11 in binary
# 2 | 3 = 10 | 11 = 11 => 3 in decimal
```

[ ]:
```
3
```

[ ]:
```
# Bitwise xor
6 ^ 5
# 6 = 110
# 5 = 101
# 110 ^ 101 = 011 = 3
```

[ ]:
```
3
```

5 Python List

So far we saw that a variable could store only one value. We can store multiple values in a variable. List helps us do that.

[ ]:
```
# List is enclosed in square brackets
days_of_week = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',␣
↪'Friday', 'Saturday']
days_of_week
```

[ ]:
```
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

You can find the number of items in the list using len()

[ ]:
```
# number of items in the list
len(days_of_week)
```

[ ]:
```
7
```

The items in the list can be of any data type.

[ ]:
```
my_list = [1, 3.0, 'Hello', True]
my_list
```

[ ]:
```
[1, 3.0, 'Hello', True]
```

5.1 Access List items

List is an ordered collection of items, i.e. each item has an index (position number) associated with it. The index always starts with zero. So, the first item in the list has index 0, second item has index 1 and so on. You can access any item in the list by its index, as shown in the example below.

[ ]:
```
# Let's access the first item in the list. Index 0.
print(days_of_week[0])
# fourth item in the list. Index 3
print(days_of_week[3])
```

```
Sunday
Wednesday
```

Negative indexing You can use negative index to access items from the end of the list. The last item in the list has the index -1, second last -2 and so on. Please see the examples.

[ ]:
```
# last item in the list. Index -1
print(days_of_week[-1])
# third last item in the list, Index -3
print(days_of_week[-3])
```

```
Saturday
Thursday
```

Range of indexes You can select more than one contiguous items in the list as a range (start_index : end_index+1). Examples shown below.

[ ]:
```
days_of_week_0_3 = days_of_week[1:4] # returns a new list with the range of␣
↪items selected
print(days_of_week_0_3) # items with indexes 1, 2, 3 will be printed. Remember␣
↪the end index 4 is not printed.
print(days_of_week) # the original list is unchanged
```

```
['Monday', 'Tuesday', 'Wednesday']
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

If you leave out the start index, the start index will start from 0.

[ ]:
```
days_of_week[:3] # indexes 0, 1, 2 will be selected
```

[ ]:
```
['Sunday', 'Monday', 'Tuesday']
```

If you leave out the end index, all the items from the start index to the end of the list is selected.

[ ]:
```
days_of_week[5:] # items 5, 6 will be selected
```

[ ]:
```
['Friday', 'Saturday']
```

You can use `in` operator to check if an item is present in the list.

[ ]:
```
'Saturday' in days_of_week
```

[ ]:
```
True
```

[ ]:
```
'Someday' in days_of_week
```

[ ]:
```
False
```

5.2 Change List items

Change item value If you assign a value to the list at the item number, you replace the old value with the new one.

[ ]:
```
days_of_week[1] = "Mon"
days_of_week
```

[ ]:
```
['Sunday', 'Mon', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

Change a range of item values

[ ]:
```
days_of_week[2:4] = ["Tue", "Wed"]
days_of_week
```

[ ]:
```
['Sunday', 'Mon', 'Tue', 'Wed', 'Thursday', 'Fri', 'Sat', 'Sat', 'Saturday']
```

If you assign more values than idexes specified in the range, the extra values will be inserted and the rest of the original values get pushed to the right.

[ ]:
```
days_of_week[5:6] = ["Fri", "Sat"]
print(days_of_week)
print(len(days_of_week)) # note that the size of the list expanded ()
```

```
['Sunday', 'Mon', 'Tue', 'Wed', 'Thursday', 'Fri', 'Sat', 'Sat', 'Saturday']
9
```

If you assign less values than the indexes specified in the range, the original values in the indexes not provided values to be replaced for will be deleted.

[ ]:
```
days_of_week[1:3] = ["Mondi"] # indexes given: 1, 2; values given "Mondi",␣
↪which goes at index 1 and original
```

```
                                # value at index 2 ("Tue") is deleted.
days_of_week
```

[ ]: ['Sunday', 'Mondi', 'Thursday', 'Fri', 'Sat', 'Sat', 'Saturday']

Insert items If you want to insert items without replacing original items, you need to use insert().

[ ]:
```
my_list = [1,2,3]
my_list.insert(1, "inserted")
my_list
```

[ ]: [1, 'inserted', 2, 3]

5.3 Add List items

To add items to the end of the list, use the append() method.

[ ]:
```
my_list = [1,2,3]
my_list.append(4)
my_list
```

[ ]: [1, 2, 3, 4]

To add items from another list, use extend() method.

[ ]:
```
list_to_add = [100, 200]
my_list.extend(list_to_add)
my_list # the added items are added to the end of the list
```

[ ]: [1, 2, 3, 4, 100, 200, 100, 200]

5.4 Remove List items

The remove() method removes the specified item provided as argument.

[ ]:
```
my_list = ["One", "Two", "Three"]
my_list.remove("Two")
my_list
```

[ ]: ['One', 'Three']

[ ]:
```
# if there are duplicates of the item to be removed only the first one of the␣
 ↪items will be removed
my_list = [1,2,3,4,2,3]
my_list.remove(2)
print(my_list)
my_list.remove(2)
print(my_list)
```

```
[1, 3, 4, 2, 3]
[1, 3, 4, 3]
```

Remove item at a specified index

You can use `pop()` method to remove an item at a specified index.

[ ]:
```
my_list = [1,2,3]
my_list.pop(1)
my_list
```

[ ]:
```
[1, 3]
```

If you do not give any argument to `pop()`, the last item in the list will be removed.

[ ]:
```
my_list = [1,2,3,4]
my_list.pop()
my_list
```

[ ]:
```
[1, 2, 3]
```

`del` keyword can also be used to remove an item at a specified index.

[ ]:
```
my_list = [1, 2, 3]
del my_list[0]
my_list
```

[ ]:
```
[2, 3]
```

`del` keyword can be used to delete the list itself.

[ ]:
```
num_list = [1,2,3]
del num_list
num_list
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-a66a283c5b9d> in <module>()
      1 num_list = [1,2,3]
      2 del num_list
----> 3 num_list

NameError: name 'num_list' is not defined
```

Clear the list

If you want to remove all the list items, you can use `clear()` method. **Think about the difference between del and clear().**

```
[ ]: my_list = [5,6,7]
     my_list.clear()
     my_list
```

```
[ ]: []
```

```
[ ]:
```

5.5 Sort List

You can sort a list alphanumerically using the `sort()` method. The default is ascending order.

```
[ ]: months_list = ["Jan", "Feb", "Mar", "Apr"]
     months_list.sort()
     months_list
```

```
[ ]: ['Apr', 'Feb', 'Jan', 'Mar']
```

```
[ ]: years_list = [1999, 1986, 2021, 1970]
     years_list.sort()
     years_list
```

```
[ ]: [1970, 1986, 1999, 2021]
```

For sorting by descending order, provide `reverse = True` argument to `sort()`

```
[ ]: num_list = [5,3,10, 15, -1, 100]
     num_list.sort(reverse=True)
     num_list
```

```
[ ]: [100, 15, 10, 5, 3, -1]
```

The `sort()` method case-sensitive by default. Capital letters are sorted before small letters.

```
[ ]: cities_list = ["Kathmandu", "london", "chicago", "new york", "Lahore"]
     cities_list.sort()
     cities_list
```

```
[ ]: ['Kathmandu', 'Lahore', 'chicago', 'london', 'new york']
```

If you want case-insensitive sort, provide the argument `key = str.lower` to `sort()` method.

```
[ ]: cities_list = ["Kathmandu", "london", "chicago", "new york", "Lahore"]
     cities_list.sort(key = str.lower)
     cities_list
```

```
[ ]: ['chicago', 'Kathmandu', 'Lahore', 'london', 'new york']
```

Reverse order You can use the reverse() method to reverse the order of items in the list. No sorting.

[ ]:
```
my_list = [3,4,1,5]
my_list.reverse()
my_list
```

[ ]:
```
[5, 1, 4, 3]
```

5.6 Make a copy of a list

You can use `copy()` method to make a copy of a list.

[ ]:
```
list_1 = [1,2,3,"hello"]
copy_of_list1 = list_1.copy()
copy_of_list1
```

[ ]:
```
[1, 2, 3, 'hello']
```

You can also copy a list using `list()` method.

[ ]:
```
my_list = [3,4,5]
my_list_copy = list(my_list)
my_list_copy
```

[ ]:
```
[3, 4, 5]
```

5.7 Join lists

You can join two lists by simply using the + operator.

[1]:
```
list_1 = [1,2,3]
list_2 = ["hi", "there"]
list_12 = list_1 + list_2 + list_1
print(list_12)
print(list_1)
```

```
[1, 2, 3, 'hi', 'there', 1, 2, 3]
[1, 2, 3]
```

You can also use the `extend()` method to add list_2 to at the end of list_1.

[ ]:
```
list_1 = [1,2,3]
list_2 = ["hi", "there"]
list_1.extend(list_2)
list_1
```

[ ]:
```
[1, 2, 3, 'hi', 'there']
```

Think about the difference between using the + operator and using the `extend()` method.

6 Python Conditionals

6.1 if Statement

If you want some python code to run only if some condition is true, you can use the `if` keyword. The format of the code is as follows:

[ ]:
```
# if <some condtion>:
#      code to run only if the condition is true
```

[ ]:
```
# example:
x = 7
if x > 5:
  print("x is greater than 5")
```

```
x is greater than 5
```

In the example above, `x > 5` is the condition, which evaluates to either `True` or `False`. You can use any of the comparison operators we learned earlier, in the condition.

Please notice the blank spaces (also called white spaces) before the start of the code right after the `if` statement. Those whitespaces are called **indentation**.

Please keep in mind that you need to indent the line or lines that you want to be part of the `if` statement. The indentation for all the lines in the `if` code block needs to be the same.

Indentation defines the scope of the `if` block. Scope will be covered in detail in the Advanced part of this course.

[ ]:
```
x = 10
if x > 10:
  print('x is greater than 10')
  print('Nothing will get printed if the condition is not True')
```

6.2 elif keyword

You can use `elif` with another condition, if the `if` condition before it failed, to see if the new condition passes.

[ ]:
```
x = 2 # also try with values x = 6, 4, 2

if x > 5:
  print('x is greater than 5')
elif x > 3:
  print('x is greater than 3')
elif x > 1:
```

```
  print('x is greater than 1')
```

```
x is greater than 1
```

6.3 else keyword

The `else` keyword catches anything not caught by the previous `if` and `elif` statements.

[ ]:
```
# The same code as above with the else keyword at the end
x = 1 # also try with values x = 6, 4, 2, 1

if x > 5:
  print('x is greater than 5')
elif x > 3:
  print('x is greater than 3')
elif x > 1:
  print('x is greater than 1')
else:
  print("x is less than or equal to 1")
```

```
x is less than or equal to 1
```

6.4 Short-hand if

You can put the `if` code block in one line as well.

[ ]:
```
x = 3
if x == 3: print("x is equal to 3")  # if code block in one line
```

```
x is equal to 3
```

6.5 Short-hand if - elif

You can put `if - else` code block in one line as well.

[ ]:
```
x = 2 # also try with x = 3
print("x is equal to 2") if x == 2 else print("x is not equal to 2")
```

```
x is not equal to 2
```

6.6 Nested if

You can put `if` statements inside `if` statements, which are called nested `if` statements.

[ ]:
```
x  = 35 # try with x = 20, 35
if x > 10:
  print("x is greater than 10")
  if x < 30:
    print("but it is less than 30")
  else:
```

```
    print("and it is greater than 30 as well")
```

```
x is greater than 10
and it is greater than 30 as well
```

#Python loops

Python has two loop commands:

- while
- for

6.7 while loop

In a `while` loop, the loop runs as long as the condition is true.

[ ]:
```
x = 0
while x < 5:
  print(x)
  x += 1 # DO NOT FORGET to increment x, otherwise the loop will run forever␣
 ↪(infinite loop)
```

```
0
1
2
3
4
```

6.7.1 break statement

The `break` statement breaks the loop and the execution exits the loop.

[ ]:
```
x = 0
while x < 10:
  print(x)
  x += 1
  if x == 4:
    break
```

```
0
1
2
3
```

6.7.2 continue statement

The `continue` statement stops the current loop iteration and continues with the new iteration.

[ ]:
```
x = 0
while x < 10:
  x += 1
```

```
  if x == 3:
    continue
  print(x)
```

```
1
2
4
5
6
7
8
9
10
```

6.8 for loop

The `for` loop is something you will be using a lot. It can iterate over a sequence (i.e. list, tuple, set, dict, string).

[ ]:
```
# iterate over a list
lst = ["one", "two", "three"]
for num in lst:
  print(num)
```

```
one
two
three
```

[ ]:
```
# iterate over a string (strings are sequence of characters)
for i in "python":
  print(i)
```

```
p
y
t
h
o
n
```

You can use `break` and `continue` just like in `while` loop.

6.9 The range() function

The `range()` function returns a sequence of numbers starting with 0 (by default) and incremented by 1 (by default) up to the number less than one provided as argument.

[ ]:
```
for i in range(6):
  print(i)
```

```
0
1
2
3
4
5
```

If two arguments are provided, then the sequence returned will start at the first argument and end at a number one less than the second argument.

[ ]:
```
for i in range(2, 7):
  print(i)
```

```
2
3
4
5
6
```

If you provide 3 arguments, then the third argument will be the increment value.

[ ]:
```
for i in range(2, 10, 3):
  print(i)
```

```
2
5
8
```

6.10 Nested loop

You can put a loop inside a loop, which is called a nested loop.

[ ]:
```
colors = ['red', 'blue', 'green']
objects = ['chair', 'table', 'house']

for i in colors:
  for j in objects:
    print(i, j)
```

```
red chair
red table
red house
blue chair
blue table
blue house
green chair
green table
green house
```

7 Python Functions

Functions are blocks of code which run only when called. Functions are not executed unless they are called. Functions most often take some input/s known as parameters and return some output.

7.1 Creating a function

You need to use the keyword `def` to create a function in Python.

[ ]:
```
# function definition
def add(x, y): # here 'add' is the function name; x and y are parameters
  return x+y # the function returns the sum of x and y

# calling the function
add(2,3) # here 2 and 3 are inputs to the function which are known as arguments.
```

[ ]:
```
5
```

Parameters vs. Arguments Parameters are the variables defined inside the parentheses right after the function name in the function definition. Arguments are the values provided to the function inside the parentheses right after the function name, when calling the function.

Functions need to be called with exactly the same number of arguments as the number of parameters defined - not fewer, not more.

[ ]:
```
add(2) # add function expects 2 arguments. However, we are passing one␣
↪argument, so get an error.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-c846fc9fb86e> in <module>()
----> 1 add(2)

TypeError: add() missing 1 required positional argument: 'y'
```

[ ]:
```
add(1,2,3) # add function expects 2 arguments. However, we are passing more␣
↪than 2 arguments, so we get an error.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-ff1b6584ceb8> in <module>()
----> 1 add(1,2,3)

TypeError: add() takes 2 positional arguments but 3 were given
```

[ ]:
```
def add(x, y):
  return x + y
```

```
add(3, 4)
```

7.2 Arbitrary arguments (*args)

If you do not know in advance how many arguments will be passed, you can define a function with arbitrary arguments by adding an * (asterisk) before the parameter name.

[ ]:
```
def add(*nums):
  sum = 0
  for i in nums:
    sum += i
  return sum

add(1,2,3,23) # try with different number of arguments
```

[ ]:
```
29
```

7.3 Default parameter value

When defining a function, we can provide a default value to parameters. When we call the function with default parameter value/s, if the argument is not passed for the default parameters, the default value is used.

[ ]:
```
def multiply(x, y=3):
  return x*y

multiply(4) # we did not pass the argument value for y, so y took the default␣
↪value of 3 as defined in the function.
```

[ ]:
```
12
```

[ ]:
```
multiply(4,5) # we provided the argument value for y, so the default value is␣
↪overridden with the value provided.
```

[ ]:
```
20
```

When defining functions with default parameter values, make sure that you put them after non-default parameters.

[ ]:
```
def subtract(x=3,y):
  return x-y

subtract(4) # returns error, since 4 is assumed to be the value for x, and␣
↪consequently it is as if y value is not provided.
```

```
  File "<ipython-input-15-928c0f31f332>", line 1
    def subtract(x=3,y):
```

```
                    ^
SyntaxError: non-default argument follows default argument
```

7.4 Lambda Functions

Lambda functions are anonymous (without a name) functions. They can take any number of arguments, but they can have only one expression. The syntax is of the following form: `lamdba` *arguments : expression*

Lambda functions are quick and compact way of creating small functions. They do not require a name, parentheses, and a return keyword as in a normal function.

[ ]:
```
sum = lambda x,y,z: x+y+z # lambda function defined and assigned to a variable␣
↪'sum'
sum(1,2,3) # 'sum' lambda function called with arguments
```

[ ]:
```
6
```

8 Python Tuples

Tuples

- can store multiple values in a single variable

- are enclosed by parentheses `()`
- are ordered collection of items and they are **immutable** (unchangeable)
- allow duplicate values

[ ]:
```
# example of a tuple
my_tuple = ("Hi", "there", 2, 3)
my_tuple
```

[ ]:
```
('Hi', 'there', 2, 3)
```

Just like a Python List, Python Tuple items are indexed starting from 0, i.e. the first item in the tuple has the index 0, the second item has the index 1 and so on.

[ ]:
```
print(my_tuple[0])
print(my_tuple[1])
print(my_tuple[2])
```

```
Hi
there
2
```

Tuple allows duplicates.

```
# duplicate items are allowed in a tuple
tuple_1 = (1, 2, "Apple", "Cherry", "Apple")
```

You can get the size of the tuple (i.e. the number of items in the tuple) using the len() method.

```
# length (number of items in tuple)
len(tuple_1)
```

```
5
```

If you need to create a tuple with only one element, you need to put a comma after the element. Otherwise, you will not get a tuple type.

```
tuple_1element = ("hello") # without a comma after the element in one-element tuple, you will not get a tuple type
type(tuple_1element)
```

```
str
```

```
tuple_1element = ("hello",) # put a comma after the element in one-element tuple
type(tuple_1element)
```

```
tuple
```

Accessing tuple items with indexing, negative indexing, range of indexes work just like in Lists.

```
my_tuple = ("Hello", "world", 1, 8.2)
print("my_tuple[0] => ", my_tuple[0]) # prints the first element of the tuple
print("my_tuple[-1] =>",my_tuple[-1]) # last element
print("my_tuple[1:3] =>",my_tuple[1:3]) # elements at index 1 (second element) up to index 2 (3rd element) - excluding index 3
print("my_tuple[-3: -1] =>",my_tuple[-3: -1]) # element at index -3 up to index -2 (excluding -1)
print("my_tuple[:2] =>",my_tuple[:2]) # elements from index 0 up to index 1
print("my_tuple[2:] =>",my_tuple[2:]) # elements from index 2 to end
```

```
my_tuple[0] =>  Hello
my_tuple[-1] => 8.2
my_tuple[1:3] => ('world', 1)
my_tuple[-3: -1] => ('world', 1)
my_tuple[:2] => ('Hello', 'world')
my_tuple[2:] => (1, 8.2)
```

You can not add items, update items, or remove items from a tuple.

You can delete a tuple with the `del` keyword.

```
del my_tuple
```

[ ]:
```
my_tuple # deleted, so will get error
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-7d63a86e361c> in <module>()
----> 1 my_tuple # deleted

NameError: name 'my_tuple' is not defined
```

8.1 Unpack Tuples

We can extract the values in tuples to variables, which is called "unpacking".

[ ]:
```
# Remember to provide as many variables as there are values in the tuple
student1, student2, student3 = ("Sam", "Harry", "Tom")
print(student1)
print(student2)
print(student3)
```

```
Sam
Harry
Tom
```

8.2 Loop through a tuple

You can conveniently loop through a tuple with a for loop.

[ ]:
```
my_tuple = ("This", "is", "great")

for item in my_tuple:
  print(item)
```

```
This
is
great
```

8.3 Join tuples

You can join tuples with a + operator.

[ ]:
```
tuple_1 = (1,2,3)
tuple_2 = (4,5,6)
tuple_3 = (7,8,9)

print(tuple_1 + tuple_2)
print(tuple_1 + tuple_2 + tuple_3)
```

```
(1, 2, 3, 4, 5, 6)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

8.4 Multiply tuples

You can multiply a tuple and an integer with a * operator.

[ ]:
```
(1,2,3) * 3
```

[ ]:
```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

9 Python Sets

Python set can store multiple values in a variable, just like List and Tuple.

Python sets are:

- unordered
- unindexed (as opposed to lists and tuples, which are indexed)
- do not allow duplicates (as opposed to lists and tuples, which allow duplicates)
- immutable (elements can not be changed, removed or updated) - allows adding of new elements
- enclosed by curly braces {}

[ ]:
```
# use curly braces {} to define a set
my_set = {2, "Hi", 5.65, True}
my_set
```

[ ]:
```
{2, 5.65, 'Hi', True}
```

[ ]:
```
# duplicates are not retained
set_1 = {1, 2, "Hi", "there", "Hi", False}
set_1
```

[ ]:
```
{1, 2, False, 'Hi', 'there'}
```

[ ]:
```
# length (number of items) of a set
len(set_1)
```

[ ]:
```
5
```

9.1 Access Set items

You can loop through a set to access the items of the set.

[ ]:
```
for i in set_1:
  print(i)
```

```
False
1
2
Hi
there
```

You can check if a particular item is in the set by using the `in` keyword.

[ ]:
```
1 in set_1
```

[ ]:
```
True
```

[ ]:
```
"Hey" in set_1
```

[ ]:
```
False
```

9.2 Add items to set

You can not change the items already in a set, however, you can add items to the set by using the add() method.

[ ]:
```
my_set = {1,2,3}
my_set.add("python")
my_set
```

[ ]:
```
{1, 2, 3, 'python'}
```

You can add items from another set by using the `update()` method.

[ ]:
```
your_set = {"Hi", "there"}
my_set.update(your_set)
my_set
```

[ ]:
```
{1, 2, 3, 'Hi', 'python', 'there'}
```

The `update()` method can take a list, a tuple, or a dictionary.

[ ]:
```
# using update() with a list
my_set = {1,2}
lst = ["three", "four"]
my_set.update(lst)
my_set
```

[ ]:
```
{1, 2, 'four', 'three'}
```

[ ]:
```
# using update() with a tuple
my_set = {1,2}
my_tup = ("three", "four")
my_set.update(my_tup)
```

```
my_set
```

[ ]: {1, 2, 'four', 'three'}

10 Python Dictionary

Python dictionary stores data in the form of key, value pairs (key:value) separated by commas and the pairs are enclosed by curly braces.

Example: ages_dict = {“Tom”: 30, “Harry”: 25, “Sarah”: 28} In the above example, “Tom”, “Harry”, and “Sarah” are keys, and 30, 25, and 28 are corresponding values.

Python dictionaries:

- are ordered (have indexes) [in Python 3.7 and later]
- are mutable (items can be added, updated, and deleted)
- do not allow duplicate keys

[ ]:
```
# Example
city_pop = {
  "Chicago": 3.5,
  "New York": 7,
  "LA": 6
}

city_pop
```

[ ]: {'Chicago': 3.5, 'LA': 6, 'New York': 7}

10.1 Access dictionary items

You can get the value of any key in the dictionary in the following way.

name_of_dict[key]

[ ]:
```
# getting the value (population) of "Chicago" (key)
city_pop["Chicago"]
```

[ ]: 3.5

You can also use the `get()` method to get the value of the key provided as argument.

[ ]:
```
city_pop.get("LA")
```

[ ]: 6

You can use the `keys()` method to get a list of all the keys in the dictionary.

[ ]:
```
city_pop.keys()
```

[ ]:
```
dict_keys(['Chicago', 'New York', 'LA'])
```

You can use the `values()` method to get a list of all the values in the dictionary.

[ ]:
```
city_pop.values()
```

[ ]:
```
dict_values([3.5, 7, 6])
```

In order to access the items in the dictionary, use the `items()` method.

[ ]:
```
# the result will be a list of tuples, each tuple containing a pair of key and␣
↪value
city_pop.items()
```

[ ]:
```
dict_items([('Chicago', 30), ('New York', 7), ('Kathmandu', 2.5), ('Pokhara',
1.5)])
```

You can check if a particular key exists in a dicionary, by using the `in` keyword.

[ ]:
```
"Chicago" in city_pop
```

[ ]:
```
True
```

[ ]:
```
"San Franscisco" in city_pop
```

[ ]:
```
False
```

10.2 Change dictionary items

If you want to update the value of a particular key, then you can just assign a new value to the key in the following way.

[ ]:
```
# dict[ key ] = new value
city_pop["LA"] = 20
city_pop
```

[ ]:
```
{'Chicago': 30, 'Kathmandu': 2.5, 'LA': 20, 'New York': 7, 'Pokhara': 1.5}
```

You can also update the value of a particular key by using the `update()` method.

[ ]:
```
city_pop.update({ "Chicago": 30 })
city_pop
```

[ ]:
```
{'Chicago': 30, 'Kathmandu': 2.5, 'LA': 20, 'New York': 7, 'Pokhara': 1.5}
```

10.3 Add items to dictionary

If you want to add items to a dictionary, you can do so by assigning a value to a new key in the following manner.

```
[ ]: city_pop["Kathmandu"] = 2.5
     city_pop
```

```
[ ]: {'Chicago': 30, 'Kathmandu': 2.5, 'LA': 20, 'New York': 7, 'Pokhara': 1.5}
```

You can also use the `update()` method to add a new item in the dictionary.

```
[ ]: city_pop.update({"Pokhara": 1.5})
     city_pop
```

```
[ ]: {'Chicago': 30, 'Kathmandu': 2.5, 'LA': 20, 'New York': 7, 'Pokhara': 1.5}
```

10.4 Remove items from dictionary

You can use the `pop()` method with key specified as argument to remove the item with the key from a dictionary.

```
[ ]: city_pop.pop("LA")
     city_pop
```

```
[ ]: {'Chicago': 30, 'Kathmandu': 2.5, 'New York': 7, 'Pokhara': 1.5}
```

The `del` keyword can also delete a specified key value pair in the following way.

```
[ ]: del city_pop["Pokhara"]
     city_pop
```

```
[ ]: {'Chicago': 30, 'Kathmandu': 2.5, 'New York': 7}
```

The `clear()` method deletes all the items in the dictionary.

```
[ ]: city_pop.clear()
     city_pop
```

```
[ ]: {}
```

You can use the `del` keyword to delete the dictionary itself from memory.

```
[ ]: del city_pop
```

```
[ ]: # city_pop is no longer accessible, as it has already been deleted.
     city_pop
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-37-ac90209871d2> in <module>()
----> 1 city_pop
```

```
NameError: name 'city_pop' is not defined
```