Summary: Integers, Floats, Lists, Dictionaries, Tuples, dir, help

In this section you learned that:

* **Integers** are for representing whole numbers:

1. rank = 10
2. eggs = 12
3. people = 3

* **Floats** represent continuous values:

1. temperature = 10.2
2. rainfall = 5.98
3. elevation = 1031.88

* **Strings** represent any text:

1. message = "Welcome to our online shop!"
2. name = "John"
3. serial = "R001991981SW"

* **Lists** represent arrays of values that may change during the course of the program:

1. members = ["Sim Soony", "Marry Roundknee", "Jack Corridor"]
2. pixel\_values = [252, 251, 251, 253, 250, 248, 247]

* **Dictionaries** represent pairs of keys and values:

1. phone\_numbers = {"John Smith": "+37682929928", "Marry Simpons": "+423998200919"}
2. volcano\_elevations = {"Glacier Peak": 3213.9, "Rainer": 4392.1}

* **Keys** of a dictionary can be extracted with:

1. phone\_numbers.keys()

* **Values** of a dictionary can be extracted with:

1. phone\_numbers.values()

* **Tuples** represent arrays of values that are not to be changed during the course of the program:

1. vowels = ('a', 'e', 'i', 'o', 'u')
2. one\_digits = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

* To find out what **attributes** a type has:

1. dir(str)
2. dir(list)
3. dir(dict)

* To find out what Python **builtin functions** there are:

1. dir(\_\_builtins\_\_)

* **Documentation** for a Python command can be found with:

1. help(str)
2. help(str.replace)
3. help(dict.values)

Summary: Positive/Negative Indexes, Slicing

In this section you learned that:

* Lists, strings, and tuples have a **positive index** system:

1. ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. 0 1 2 3 4 5 6

* And a **negative index** system:

1. ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. -7 -6 -5 -4 -3 -2 -1

* In a list, the **2nd**, **3rd**, and **4th** items can be accessed with:

1. days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. days[1:4]
3. Output: ['Tue', 'Wed', 'Thu']

* **First three items of a list**:

1. days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. days[:3]
3. Output:['Mon', 'Tue', 'Wed']

* **Last three items of a list**:

1. days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. days[-3:]
3. Output: ['Fri', 'Sat', 'Sun']

* **Everything but the last**:

1. days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. days[:-1]
3. Output: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']

* **Everything but the last two**:

1. days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
2. days[:-2]
3. Output: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']

* A single in a **dictionary** can be accessed using its key:

1. phone\_numbers = {"John Smith":"+37682929928","Marry Simpons":"+423998200919"}
2. phone\_numbers["Marry Simpsons"]
3. Output: '+423998200919'

Bonus Code: Using "and" and "or" in a Conditional

You learned to check for one single condition:

1. x = 1
3. if x == 1:
4. print("Yes")
5. else:
6. print("No")

You can also check if two conditions are met at the same time using an and operator:

1. x = 1
2. y = 1
4. if x == 1 and y==1:
5. print("Yes")
6. else:
7. print("No")

That will return Yes since x == 1 and y ==1 are both True.

You can also check if one of two conditions are met using an or operator:

1. x = 1
2. y = 1
4. if x == 1 or y==2:
5. print("Yes")
6. else:
7. print("No")

That will return Yes since at least one of the conditions is True. In this case x == 1 is True.

Summary: Functions and Conditionals

In this section you learned to:

* Define a **function**:

1. def cube\_volume(a):
2. return a \* a \* a

* Write a **conditional** block:

1. message = "hello there"
3. if "hello" in message:
4. print("hi")
5. else:
6. print("I don't understand")

* Write a conditional block of **multiple conditions**:

1. message = "hello there"
3. if "hello" in message:
4. print("hi")
5. elif "hi" in message:
6. print("hi")
7. elif "hey" in message:
8. print("hi")
9. else:
10. print("I don't understand")

* Use the and operator to check if **both conditions** are True at the same time:

1. x = 1
2. y = 1
4. if x == 1 and y==1:
5. print("Yes")
6. else:
7. print("No")

Output is Yes since both x and y are 1.

* Use the or operator to check if **at least one condition** is True:

1. x = 1
2. y = 2
4. if x == 1 or y==2:
5. print("Yes")
6. else:
7. print("No")

Output is Yes since x is 1.

* Check if a value is of a certain **type** with:

1. isinstance("abc", str)
2. isinstance([1, 2, 3], list)

or

1. type("abc") == str
2. type([1, 2, 3]) == lst

Summary: Processing User Input

In this section you learned that:

* A Python program can get **user input** via the input function:
* The **input** **function** halts the execution of the program and gets text input from the user**:**

1. name = input("Enter your name: ")

* The input function converts any **input to a string**, but you can convert it back to int or float:

1. experience\_months = input("Enter your experience in months: ")
2. experience\_years = int(experience\_months) / 12

* You can **format strings** with (works both on Python 2 and 3):

1. name = "Sim"
2. experience\_years = 1.5
3. print("Hi %s, you have %s years of experience." % (name, experience\_years))

Output: Hi Sim, you have 1.5 years of experience.

* You can also **format strings** with (Python 3 only):

1. name = "Sim"
2. experience\_years = 1.5
3. print("Hi {}, you have {} years of experience".format(name, experience\_years))

Output: Hi Sim, you have 1.5 years of experience.

For Loop Over a Function

Note that using loops you can call any function multiple times, even your own functions. Let's suppose we defined this function:

1. def celsius\_to\_kelvin(cels):
2. return cels + 273.15

That is a function that gets a number as input, adds 273.15 to it and returns the result. A *for* loop allows us to execute that function over a list of numbers:

1. monday\_temperatures = [9.1, 8.8, -270.15]
3. for temperature in monday\_temperatures:
4. print(celsius\_to\_kelvin(temperature))

The output of that would be:

282.25  
281.95  
3.0

So, in the first iteration celsius\_to\_kelvin(9.1) was executed, in the second celsius\_to\_kelvin(8.8) and in the third celsius\_to\_kelvin(-270.15).

That's just something to keep in mind.

Bonus Code: Dictionary Loop and String Formatting

You can combine a dictionary for loop with string formatting to create text containing information from the dictionary:

1. phone\_numbers = {"John Smith": "+37682929928", "Marry Simpons": "+423998200919"}
3. for pair in phone\_numbers.items():
4. print("{} has as phone number {}".format(pair[0], pair[1]))

Another (better) way to do it::

1. phone\_numbers = {"John Smith": "+37682929928", "Marry Simpons": "+423998200919"}
3. for key, value in phone\_numbers.items():
4. print("{} has as phone number {}".format(key, value))

In both cases the output is:

Output:

John Smith has as phone number +37682929928

Marry Simpons has as phone number +423998200919

Summary: Loops

In this section you learned that:

* **For loops** are useful for executing a command over a large number of items.
* You can create a **for loop** like so:

1. for letter in 'abc':
2. print(letter.upper())

Output:

A  
B  
C

* The name after for (e.g. letter) is just a variable name
* You can loop over **dictionary keys**:

1. phone\_numbers = {"John Smith":"+37682929928","Marry Simpons":"+423998200919"}
2. for value in phone\_numbers.keys():
3. print(value)

Output:

John Smith  
Marry Simpsons

* You can loop over **dictionary values**:

1. phone\_numbers = {"John Smith":"+37682929928","Marry Simpons":"+423998200919"}
2. for value in phone\_numbers.values():
3. print(value)

Output:

+37682929928  
+423998200919

* You can loop over **dictionary items**:
  1. phone\_numbers = {"John Smith":"+37682929928","Marry Simpons":"+423998200919"}
  2. for key, value in phone\_numbers.items():
  3. print(key, value)

Output:

('John Smith', '+37682929928')

('Marry Simpons', '+423998200919')

* **While loops** will run as long as a condition is true:
  1. while datetime.datetime.now() < datetime.datetime(2090, 8, 20, 19, 30, 20):
  2. print("It's not yet 19:30:20 of 2090.8.20")

The loop above will print out the string inside print() over and over again until the 20th of August, 2090.

Summary: List Comprehensions

In this section you learned that:

* A list comprehension is an expression that creates a list by iterating over another container.
* A **basic** list comprehension:
  1. [i\*2 for i in [1, 5, 10]]

Output: [2, 10, 20]

* List comprehension with **if** condition:
  1. [i\*2 for i in [1, -2, 10] if i>0]

Output: [2, 20]

* List comprehension with an **if** **and** **else** condition:
  1. [i\*2 if i>0 else 0 for i in [1, -2, 10]]

Output: [2, 0, 20]

Summary: More on Functions

In this section you learned that:

* Functions can have more than one **parameter**:

1. def volume(a, b, c):
2. return a \* b \* c

* Functions can have **default** parameters (e.g. coefficient):

1. def converter(feet, coefficient = 3.2808):
2. meters = feet / coefficient
3. return meters
5. print(converter(10))

Output: 3.0480370641306997

Arguments can be passed as **non-keyword** (positional) arguments (e.g. a) or **keyword** arguments (e.g. b=2 and c=10):

1. def volume(a, b, c):
2. return a \* b \* c
4. print(volume(1, b=2, c=10))

* An **\*args** parameter allows the  function to be called with an arbitrary number of non-keyword arguments:

1. def find\_max(\*args):
2. return max(args)
3. print(find\_max(3, 99, 1001, 2, 8))

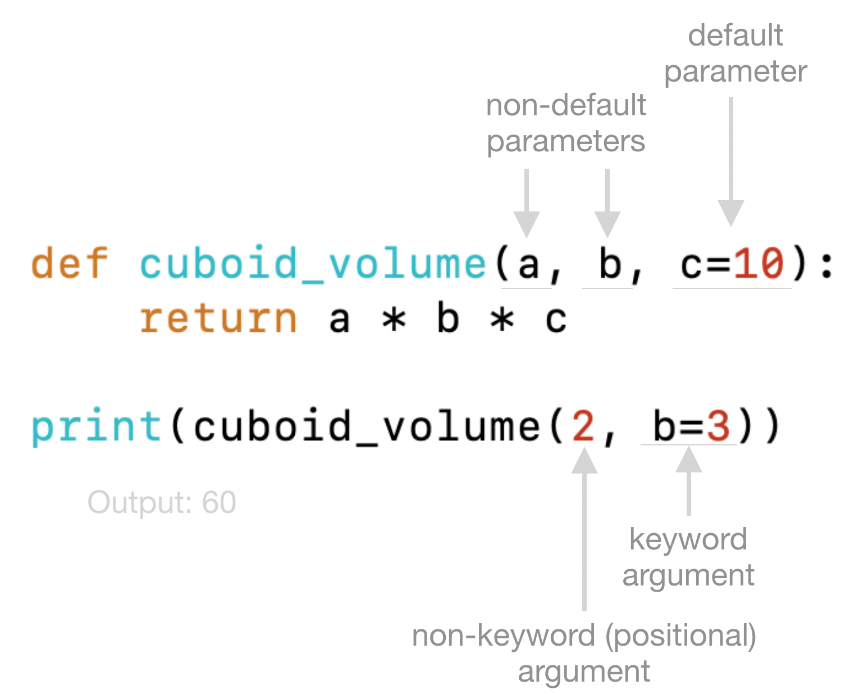
Output: 1001

* An **\*\*kwargs** parameter allows the function to be called with an arbitrary number of keyword arguments:

1. def find\_winner(\*\*kwargs):
2. return max(kwargs, key = kwargs.get)
4. print(find\_winner(Andy = 17, Marry = 19, Sim = 45, Kae = 34))

Output: Sim

* Here's a summary of function elements:



Summary: File Processing

In this section you learned that:

* You can **read** an existing file with Python:

1. with open("file.txt") as file:
2. content = file.read()

* You can **create** a new file with Python and **write** some text on it:

1. with open("file.txt", "w") as file:
2. content = file.write("Sample text")

* You can **append** text to an existing file without overwriting it:

1. with open("file.txt", "a") as file:
2. content = file.write("More sample text")

* You can both **append and read** a file with:

1. with open("file.txt", "a+") as file:
2. content = file.write("Even more sample text")
3. file.seek(0)
4. content = file.read()

Summary: Imported Modules

In this section you learned that:

* **Builtin objects** are all objects that are written inside the Python interpreter in C language.
* **Builtin modules** contain builtins objects.
* Some builtin objects are not immediately available in the global namespace. They are parts of a builtin module. To use those objects the module needs to be **imported** first. E.g.:
  1. import time
  2. time.sleep(5)
* **A list of all builtin modules** can be printed out with:
  1. import sys
  2. sys.builtin\_module\_names
* **Standard libraries** is a jargon that includes both builtin modules written in C and also modules written in Python.
* **Standard libraries** written in Python reside in the Python installation directory as *.py* files. You can find their directory path with sys.prefix.
* **Packages** are a collection of *.py* modules.
* **Third-party libraries** are packages or modules written by third-party persons (not the Python core development team).
* Third-party libraries can be **installed** from the terminal/command line:

Windows:

pip install pandas

Mac and Linux:

pip3 install pandas

More SQL Statements

In the example you just saw we used the following SQL statement in our Python code:

1. query = cursor.execute("SELECT \* FROM Dictionary WHERE Expression = 'rain'")

That statement retrieved all the rows of the *Dictionary* table where the value of the column *Expression* was *rain*. The string inside *cursor.execute()* is SQL code that Python sends to the database. That kind of language is understood by the database.

Here are some more examples of SQL queries that you can try out from within your Python script just like we did previously:

* Get all rows where the value of the column Expression starts with *r:*

1. "SELECT \* FROM Dictionary WHERE Expression  LIKE 'r%'"

* Get all rows where the value of the column Expression starts with *rain:*

1. "SELECT \* FROM Dictionary WHERE Expression  LIKE 'rain%'"

* All rows where the length of the value of the column Expression is less than four characters:

1. "SELECT \* FROM Dictionary WHERE length(Expression) < 4"

* All rows where the length of the value of the column Expression is four characters:

1. "SELECT \* FROM Dictionary WHERE length(Expression) = 4"

* All rows where the length of the value of the column Expression is greater than 1 but less than 4 characters:

1. "SELECT \* FROM Dictionary WHERE length(Expression) > 1 AND length(Expression) < 4"

* All rows of column *Definition* where the value of the column Expression starts with *r:*

1. "SELECT Definition FROM Dictionary WHERE Expression LIKE 'r%'"

Make sure you have pandas installed. You can install it with pip:

pip install pandas

or

pip3 install pandas

Also, in the next lecture, we will use an enhanced Python interactive shell called IPython.

IPython is just like the normal shell you get when you run python, but IPython provides better printing for large text. This ability makes IPython suitable for data analysis because the program prints data in a well-structured format. You can install IPython with pip:

pip install ipython

or

pip3 install ipython