Python MOOC Course – University of Helsinki

Part 01

Printing

Use **print**("Text").

# for comment

If print("2+10+20") ottieni → 2+10+20

If print(2*2) ottieni → 4

Normalmente il print stampa la stringa seguita dal carattere di change of line. Ciò si può cambiare inserendo il parametro end come segue:

print("Ciao", end="") così non andrà a capo con il prossimo print.

Getting Information From the User

Si sfrutta il commando "**input**" che legge in input una riga scritta dall'utente

Come negli altri linguaggi di programmazione, il **+ concatena** più stringhe.

Variable

```
result = 10 * 25
# the following line produces an error
print("The result is " + result)
```

Le seguenti righe di codice producono un errore perché result è un intero, mentre il resto è una stringa (in python da errore). Quindi la soluzione è **castare** in string anche result

```
result = 10 * 25
print("The result is " + str(result))
```

Oppure un'altra soluzione **built-in** è la seguente (separare le cose da stampare con una **virgola**, così vengono stampate indipendentemente dal tipo. Viene anche aggiunto in automatico uno spazio bianco tra i diversi elementi da stampare)

```
result = 10 * 25
print("The result is", result)
```

f-strings è un altro metodo di formattazione in python

```python
result = 10 * 25
print(f"The result is {result}")
```

Il primo carattere del print che è una **f** indica a python che stiamo usando una f-string. Le variabili verranno poi inserite all'interno delle parentesi graffe. Questo rende più semplice la scrittura nel momento in cui abbiamo tante variabili.

Funzionano da python 3.6 in poi.

Arithmetic operation

Operator	Purpose	Example	Result
`+`	Addition	`2 + 4`	`6`
`-`	Subtraction	`10 - 2.5`	`7.5`
`*`	Multiplication	`-2 * 123`	`-246`
`/`	Division (floating point result)	`9 / 2`	`4.5`
`//`	Division (integer result)	`9 // 2`	`4`
`%`	Modulo	`9 % 2`	`1`
`**`	Exponentiation	`2 ** 3`	`8`

Con la **divisione //,** il risultato è arrotondato per difetto.

Se un'operazione ha un operando intero e uno float, il risultato sarà automaticamente in float (**domina** come tipo)

Se vogliamo prendere in input un intero, lo prendiamo come stringa e lo **castiamo** a **int** (int()), come prima con le stringa per la formattazione dell'output.

```python
year = int(input("Which year were you born? "))
print(f"Your age at the end of the year 2021: {2021 - year}" )
```

In modo simile esiste anche la conversione in **float** tramite float().

Conditional Statement

```python
number = int(input("Please type in a number: "))
```

```
if number < 0:
    print("The number is negative.")

if number > 0:
    print("The number is positive.")

if number == 0:
    print("The number is zero.")
```

Da questo pezzo di codice si nota la sintassi dei **Conditional Statement**, cioè si usano i : e non si usano le parentesi tonde.

Inoltre, python riconosce questi conditional statement o comunque i cicli in generale grazie all'indentazione. Quindi riconosce il corpo di un blocco di esecuzione tramite l'indentazione.

Per le condizioni si può usare direttamente la keyword True o False.

Part 02

Programming Terminology

Si può usare la funzione **type(nome_var)** che stampa il tipo della variabile passata come parametro

La funzione **len** può essere usata per trovare la lunghezza di una stringa

More Conditionals

```
If cond:
        statement
elif cond:
        statement
else:
        statement
```

Combining Conditions

In python si usa “and”, “or” e “not” scritti letteralmente. Nel caso in cui volessimo controllare se un valore cade all’interno di un certo intervallo, invece che usare and, possiamo scrivere a <= x <= b

Simple Loops

```
while True:
    number = int(input("Please type in a number, -1 to quit: "))

    if number == -1:
        break

    print(number ** 2)

print("Thanks and bye!")
```

Part 03

Loops with conditions

```
while <condition>:
    <block>
```

Working with strings

```
word = "banana"
print(word*3)
```

Si può fare per far stampare n volte la parola.

The function **len** returns the number of characters in a string, which is always an integer value.

As strings are essentially sequences of characters, any single character in a string can also be retrieved. The **operator []** finds the character with the index specified within the brackets.

You can also use negative **indexing** to access characters counting from the end of the string. The last character in a string is at index -1, the second to last character is at index -2, and so forth

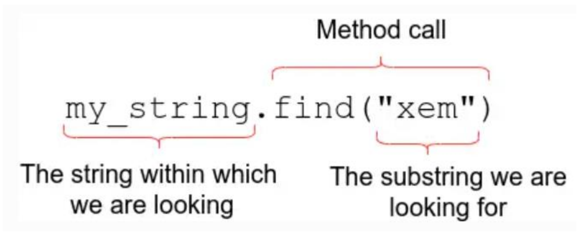
In Python programming, the process of selecting **substrings** is usually called slicing, and a substring is often referred to as a slice of the string.

[a:b] → This means the slice begins at the index a and ends at the last character before index b - that is, including the first, but excluding the last.

The **in operator** can tell us if a string contains a particular substring.

The **operator in** returns a **Boolean value**, so it will only tell us if a substring exists in a string, but it will not be useful in finding out where exactly it is.

Instead, the Python **string method find** can be used for this purpose. It takes the substring searched for as an argument, and returns either the first index where it is found, or -1 if the substring is not found within the string.


Method call
my_string.find("xem")
The string within which we are looking
The substring we are looking for

More Loops

N/A

Part 04

Defining Functions

Any function definition begins with the keyword **def**, short for **define**. Then comes the name of the **function**, followed by **parentheses** and a **colon** character. This is called the **header** of the function. After this, **indented** just like while and if blocks, comes the **body** of the function.

```
def message():
    print("This is my very own function!")


message()
```

Le chiamate a queste funzioni vanno poi inserite nel main del programma

```
def greet():
    print("Hi!")


# Write your main function within a block like this:
```

```python
if __name__ == "__main__":
    greet()
```

Functions often take one or more **arguments**, which may affect what the function does.

The functions you define yourself can also return values. To do this you need the **return statement**.

Per evitare problem di tipi nei parametri delle funzioni, you can include type hints in your function definitions. The **type hint** specifies the type of the argument intended for the function:

```python
def print_many_times(message : str, times : int):
    while times > 0:
        print(message)
        times -= 1
```

Similarly, the return value of a function can be hinted at in the function definition:

```python
def ask_for_name() -> str:
    name = input("Mikä on nimesi? ")
    return name
```

Lists

A Python **list is a collection of values** which is accessed via a single variable name.

```python
my_list = [7, 2, 2, 5, 2]

print(my_list[0])
print(my_list[1])
print(my_list[3])

print("The sum of the first two items:", my_list[0] + my_list[1])
```

The entire contents of the list can also be **printed out**:

```python
my_list = [7, 2, 2, 5, 2]
print(my_list)
```

The **append method** adds items to the end of a list. It works like this:

```
numbers = []
numbers.append(5)
numbers.append(10)
numbers.append(3)
print(numbers)
```

If you want to specify a location in the list where an item should be added, you can use the **insert method**.

```
numbers = [1, 2, 3, 4, 5, 6]
numbers.insert(0, 10)
print(numbers)
numbers.insert(2, 20)
print(numbers)
```

There are two different approaches to **removing an item** from a list:

- If the index of the item is known, you can use the **method pop**.
- If the contents of the item are known, you can use the **method remove**.

So, the **method pop** takes the index of the item you want to remove as its argument. The following program removes items at indexes 2 and 3 from the list. Notice how the indexes of the remaining items change when one is removed.

It's useful to remember that the method pop also returns the removed item.

The **method remove**, on the other hand, takes the value of the item to be removed as its argument.

The method removes the first occurrence of the value in the list, much like the string function find returns the first occurrence of a substring

Just like with strings, we can check for the presence of an item with the **in operator.**

The items in a list can be sorted from smallest to greatest with the **method sort**. Notice how the method modifies the list itself. Sometimes we don't want to change the original list, so **we use the function sorted instead. It returns a sorted list**.

The **functions max and min**, short for maximum and minimum, return the greatest and smallest item in a list, respectively. The **function sum** returns the sum of all items in a list.

Our own functions can also take a list as an argument and produce a list as a return value. The following function works out the central value in an ordered list, also called the median value:

```
def median(my_list: list):
    ordered = sorted(my_list))
    list_centre = len(ordered) // 2
    return ordered[list_centre]
```

Definite Iteration

When using a **while** loop the program doesn't "know" beforehand how many iterations the loop will perform. It will repeat until the condition becomes false, or the loop is otherwise broken out of. That is why it falls under indefinite iteration. With a **for loop** the number of iterations is determined when the loop is set up, and so it falls under definite iteration.

```
for <variable> in <collection>:
    <block>
```

Often you know how many times you want to repeat a certain bit of code. You might, for example, wish to go through all the numbers between 1 and 100. The **range function** plugged into a for loop will do this for you. The simplest way is to give the function just one argument, which signifies the end-point of the range. The **end-point** itself is **excluded**, just like with string slices. In other words, the function call range(n) provides a loop with a range from 0 to n-1

```
for i in range(5):
    print(i)
```

With two arguments, the function will return a range between the two numbers. The function **range(a,b)** provides a range starting from a and ending at b-1:

```
for i in range(3, 7):
    print(i)
```

The function call **range(a, b, c)** provides a range starting from a, ending at b-1, and changing by c with every step

```
for i in range(1, 9, 2):
    print(i)
```

Then the range will be in reversed orded:

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

The **function range** returns a **range object**, which in many ways behaves like a list, but isn't actually one.

```
numbers = range(2, 7)
```

The **function list** will convert a range into a list. The list will contain all the values that are in the range.

```
numbers = list(range(2, 7))
```

Print statement formatting

```
number = 1/3
print(f"The number is {number:.2f}")
```

The format specifier .2f states that we want to display 2 decimals. The letter f at the end means that we want the variable to be displayed as a float, i.e. a floating point number.

Here's another example, where we specify the amount of whitespace reserved for the variable in the printout. Both times the variable name is included in the resulting string, it has a space of 15 characters reserved. First the names are justified to the left, and then they are justified to the right:

```
names =  [ "Steve", "Jean", "Katherine", "Paul" ]
for name in names:
  print(f"{name:15} centre {name:>15}")
```

```
Steve           centre           Steve
Jean            centre            Jean
Katherine       centre       Katherine
Paul            centre            Paul
```

The uses of f-strings are not restricted to print commands. They can be assigned to variables and combined with other strings:

```
name = "Larry"
age = 48
city = "Palo Alto"
greeting = f"Hi {name}, you are {age} years of age"
print(greeting + f", and you live in {city}")
```

More Strings and Lists

Lists can be sliced just like strings:

```
my_list = [3,4,2,4,6,1,2,4,2]
print(my_list[3:7])
```

In fact, **the [] syntax** works very similarly to the range function, which means we can also give it a step:

```
my_string = "exemplary"
print(my_string[0:7:2])
my_list = [1,2,3,4,5,6,7,8]
print(my_list[6:2:-1])
```

The **method count** counts the number of times the specified item or substring occurs in the target. The method works similarly with both strings and lists:

```
my_string = "How much wood would a woodchuck chuck if a woodchuck could
chuck wood"
print(my_string.count("ch"))


my_list = [1,2,3,1,4,5,1,6]
print(my_list.count(1))
```

The **method replace** creates a new string, where a specified substring is replaced with another string:

```
my_string = "Hi there"
new_string = my_string.replace("Hi", "Hey")
print(new_string)
```

Part 05

More Lists

The items in a list can be lists themselves:

```
my_list = [[5, 2, 3], [4, 1], [2, 2, 5, 1]]
print(my_list)
print(my_list[1])
print(my_list[1][0])
```

Remember that lists can contain items of different types. You could store information about a person in a list. For instance, you could include their name as the first item, their age as the second item, and their height in meters as the third item

A two-dimensional array, or a matrix, is also a natural application of a list within a list. Since a **matrix** is a list containing lists, the individual elements within the matrix can be accessed using consecutive square brackets. The first index refers to the row, and the second to the column. Indexing starts from zero. The matrix is stored by rows.

References

What is stored in a variable is not the value per se, but a reference to the object which is the actual value of the variable. The object can be e.g. a number, a string or a list.

So, a reference tells us where the value can be found. The **function id** can be used to find out the exact location the variable points to:

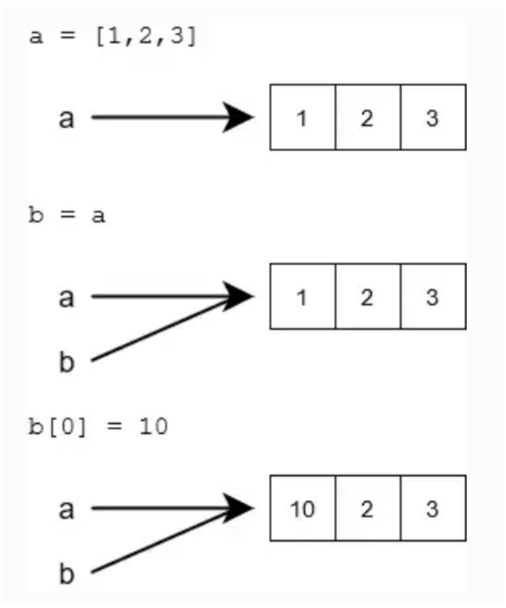
```
a = [1, 2, 3]
print(id(a))
b = "This is a reference, too"
print(id(b))
```

The **reference**, or the ID of the variable, is an integer, which can be thought of as the **address** in computer memory where the value of the variable is stored.

It may surprise you that also the **basic data types** int, float and bool are **immutable** in Python.

The assignment b = a copies the value stored in variable a to the variable b. However, the value stored in a is not the list itself, but a reference to the list.

So, the assignment b = a copies the reference. As a result there are now two references to the same memory location containing the list.



The **notation [:]** selects all items in the collection. As a side effect, it creates a copy of the list:

```
my_list = [1,2,3,4]
new_list = my_list[:]
```

When you pass a list as an **argument** to a function, you are passing a **reference** to that list. This means that the function can modify the list **directly**.

It is generally considered a good programming practice to avoid **causing side effects** with functions. Side effects can make it more difficult to verify that the program functions as intended in all situations.

Functions free of side effects are also called **pure functions**.

Dictionary

Another central data structure in Python is the **dictionary**. In a dictionary, the items are **indexed by keys**. Each key maps to a value. The values stored in the dictionary can be accessed and changed using the key.

```
my_dictionary = {}

my_dictionary["apina"] = "monkey"
my_dictionary["banaani"] = "banana"
my_dictionary["cembalo"] = "harpsichord"
```

The **notation {}** creates an empty dictionary, to which we can now add content.

L'operatore in, usato con I dizionari, controlla se l'elemento è presente nelle chiavi del dizionario, non nei valori associati

The data type is called dictionary, but it does not have to contain only strings. For example, in the following dictionary the keys are strings, but the values are integers:

```
results = {}
results["Mary"] = 4
results["Alice"] = 5
results["Larry"] = 2
```

Each key can appear only once in the dictionary. If you add an entry using a key that already exists in the dictionary, the original value mapped to that key is replaced with the new value.

The familiar **for item in collection** loop can be used to traverse a dictionary, too. When used on the dictionary directly, the loop goes through the keys stored in the dictionary, one by one.

```
my_dictionary = {}

my_dictionary["apina"] = "monkey"
my_dictionary["banaani"] = "banana"
my_dictionary["cembalo"] = "harpsichord"

for key in my_dictionary:
    print("key:", key)
    print("value:", my_dictionary[key])
```

Sometimes you need to traverse the entire contents of a dictionary. **The method items** returns all the keys and values stored in the dictionary, one pair at a time:

```
for key, value in my_dictionary.items():
    print("key:", key)
    print("value:", value)
```

It is naturally possible to also **remove key-value paris** from the dictionary. There are two ways to accomplish this. The first is the **command del**:

```
staff = {"Alan": "lecturer", "Emily": "professor", "David": "lecturer"}
del staff["David"]
print(staff)
```

The other way to delete entries in a dictionary is **the method pop**

```
staff = {"Alan": "lecturer", "Emily": "professor", "David": "lecturer"}
deleted = staff.pop("David")
print(staff)
print(deleted, "deleted")
```

By default, pop will also cause an error if you try to delete a key which is not present in the dictionary. It is possible to avoid this by giving the **method a second argument**, which contains a default return value. This value is returned in case the key is not found in the dictionary. The special Python value None will work here:

```
staff = {"Alan": "lecturer", "Emily": "professor", "David": "lecturer"}
deleted = staff.pop("Paul", None)
if deleted == None:
  print("This person is not a staff member")
else:
  print(deleted, "deleted")
```

Per cancellare l'intero dizionario, non si può usare un for o si riceve un errore ("Dictionary changed size during iteration").

Per questo esiste un metodo per pulire l'intero dizionario chiamato **clean.**

```
staff.clear()
```

Tuple

Tuple is a data structure which is, in many ways, similar to a list. The most important differences between the two are:

- Tuples are enclosed in parentheses (), while lists are enclosed in square brackets []
- Tuples are immutable, while the contents of a list may change

The following bit of code creates a tuple containing the coordinates of a point:

```
point = (10, 20)
```

The items stored in a tuple are accessed by index, just like the items stored in a list:

```
point = (10, 20)
print("x coordinate:", point[0])
print("y coordinate:", point[1])
```

Because tuples are immutable, unlike lists, they can be used as **keys in a dictionary**. The following bit of code creates a dictionary, where the keys are coordinate points:

```
points = {}
points[(3, 5)] = "monkey"
points[(5, 0)] = "banana"
```

```
points[(1, 2)] = "harpsichord"
print(points[(3, 5)])
```

Attempting a similar dictionary definition using lists **would not work.**

The parentheses are not strictly necessary when defining tuples. This means we can also easily return multiple values using tuples. Let's have alook at he following example:

```
def minmax(my_list):
  return min(my_list), max(my_list)

my_list = [33, 5, 21, 7, 88, 312, 5]

min_value, max_value = minmax(my_list)
print(f"The smallest item is {min_value} and the greatest item is 
{max_value}")
```

Using parentheses may make the notation more clear. On the left hand side of the assignment statement we also have a tuple, which contains two variable names. The values contained within the tuple returned by the function are assigned to these two variables.

```
(min_value, max_value) = minmax(my_list)
```

The method **my_dictionary.items()** returns each key-value pair as a tuple, where the first item is the key and the second item is the value.

Part 06

Reading Files

A simple way to include files in a Python program is to use the **with statement**. The header line opens the file, and the block where the file can be accessed follows. After the block the file is automatically closed, and can no longer be accessed.

So, the following code opens the file, reads the contents, prints them out, and then closes the file:

```
with open("example.txt") as new_file:
    contents = new_file.read()
    print(contents)
```

The variable **new_file above is a file handle**. Through it the file can accessed while it is still open. Here we used the **method read**, which returns the contents of the file as a single string.

The read method is useful for printing out the contents of the entire file, but more often we will want to go through the file line by line.

Text files can be thought of **as lists of strings**, each string representing a single line in the file. We can go through the list with a **for loop**.

The following example reads our example file using a for loop, removes line breaks from the end of each line, counts the number of lines, and prints each line with its line number. It also keeps track of the length of the lines:

```
with open("example.txt") as new_file:
    count = 0
    total_length = 0

    for line in new_file:
        line = line.replace("\n", "")
        count += 1
        print("Line", count, line)
        length = len(line)
        total_length += length

print("Total length of lines:", total_length)
```

Reading CSV Files

CSV files are commonly used to store records of different kinds. Many database and spreadsheet programs, such as Excel, can import and export data in CSV format, which makes data exchange between different systems easy.

Python has a **string method split** for just this purpose. The method takes the separator character(s) as a string argument, and returns the contents of the target string as a list of strings, separated at the separator.

```
text = "monkey,banana,harpsichord"
words = text.split(",")
for word in words:
    print(word)
```

If we want to access the contents in the second for loop, we will have to open the file a second time:

```python
with open("people.csv") as new_file:
    # print out the names
    for line in new_file:
        parts = line.split(";")
        print("Name:", parts[0])

with open("people.csv") as new_file:
    # find the oldest
    age_of_oldest = -1
    for line in new_file:
        parts = line.split(";")
        name = parts[0]
        age = int(parts[1])
        if age > age_of_oldest:
            age_of_oldest = age
            oldest = name
    print("the oldest is", oldest)
```

While the above code would work, it contains unnecessary repetition. It is usually best to read the file just once, and store its contents in an appropriate format for further processing:

```python
people = []
# read the contents of the file and store it in a list
with open("people.csv") as new_file:
    for line in new_file:
        parts = line.split(";")
        people.append((parts[0], int(parts[1]), parts[2]))

# print out the names
for person in people:
    print("Name:", person[0])

# find the oldest
age_of_oldest = -1
for person in people:
    name = person[0]
    age = person[1]
    if age > age_of_oldest:
        age_of_oldest = age
        oldest = name
print("the oldest is", oldest)
```

We have already used the replace method to remove extra whitespace, but a more efficient solution is to use the Python **string method strip**, which removes whitespace from the beginning and end of a string. It removes all spaces, line breaks, tabs and other characters which would not normally be printed out.

```
>>> "\n\ntest\n".strip()
'test'
```

Creating a new file

If you want to create a new file, you would call **the open function with the additional argument w**, to signify that the file should be opened in write mode.

NB: if the file already exists, all the contents will be overwritten. It pays to be very careful when creating new files.

With the file open you can write data to it. You can use the method write, which takes the string that is to be written as its argument.

```
with open("new_file.txt", "w") as my_file:
    my_file.write("Hello there!")
```

If you want to **append data to the end of a file**, instead of overwriting the entire file, you should open the file in append mode with the **argument a**.

If the file doesn't yet exist, append mode works exatly like write mode.

CSV files can be written line by line with the write method just like any other file. The following example creates the file coders.csv, with each line containing the name, working environment, favourite language and years of experience of a single programmer. The fields are separated by a semicolon.

```
with open("coders.csv", "w") as my_file:
    my_file.write("Eric;Windows;Pascal;10\n")
    my_file.write("Matt;Linux;PHP;2\n")
    my_file.write("Alan;Linux;Java;17\n")
    my_file.write("Emily;Mac;Cobol;9\n")
```

Sometimes it is necessary to **clear the contents of an existing file**. Opening the file in write mode and closing the file immediately will achieve just this:

```
with open("file_to_be_cleared.txt", "w") as my_file:
    pass
```

Now the with block only contains the command pass, which doesn't actually do anything. Python does not allow empty blocks, so the command is necessary here.

It is possible to also **bypass the with block by using the following oneliner**:

```
open('file_to_be_cleared.txt', 'w').close()
```

You can also **delete a file entirely**. We will have to ask for help from the operating system to achieve this:

```
# the command to delete files is in the os module
import os

os.remove("unnecessary_file.csv")
```

Handling Errors

The are two basic categories of errors that come up in programming contexts:

- **Syntax errors**, which prevent the execution of the program
- **Runtime errors**, which halt the execution

Errors in category 1 are usually easy to fix, as the Python interpreter flags the error location when attempting to execute the program. Common syntax errors include a missing colon at the end of a header line, or a missing quotation mark at the end of a string.

Errors in category 2 can be harder to spot, as it may happen that they only occur at a certain point in the execution of a program, and only in certain circumstances. The program may work just fine in most situations, but halt due to an error in a specific marginal case. We will now concentrate on handling these types of errors.

Errors that occur while the program is already running are called **exceptions**. It is possible to prepare for exceptions, and handle them so that the execution continues despite them occurring.

Exception handling in Python is accomplished **with try and except statements**. The idea is that if something within a try block causes an exception, Python checks if there is a corresponding except block. If such a block exists, it is executed and the program themn continues as if nothing happened.

```
try:
    age = int(input("Please type in your age: "))
except ValueError:
    age = -1

if age >= 0 and age <= 150:
    print("That is a fine age")
else:
    print("This is not a valid age")
```

We can use the try block to flag that the code within the block may cause an error. In the except statement directly after the block the relevant error is mentioned

There may be more than one except block attached to each try block. For example, the following program can handle both a FileNotFoundException and a PermissionError:

```
try:
    with open("example.txt") as my_file:
        for line in my_file:
            print(line)
except FileNotFoundError:
    print("The file example.txt was not found")
except PermissionError:
    print("No permission to access the file example.txt")
```

Sometimes it is not necessary to specify the error the program prepares for. Especially when dealing with files, it is often enough to know that an error has occurred, and safely exit the program

You can also raise exceptions, with the **command raise**. It may seem like an odd idea to purposefully cause errors in your programs, but it can, in fact, be a very useful mechanism. For instance, it can sometimes be a good idea to raise an error when detecting invalid parameters. So far we have usually printed out messages when validating input, but if we are writing a function which is executed from elsewhere, just printing something out can go unnoticed when the function is called. Raising an error can make debugging easier.

```
def factorial(n):
    if n < 0:
        raise ValueError("The input was negative: " + str(n))
    k = 1
    for i in range(2, n + 1):
        k *= i
    return k

print(factorial(3))
print(factorial(6))
print(factorial(-1))
```

Part 07

Debubbing Revisited

In Python version 3.7 brought yet another easy and useful tool for debugging programs: the **breakpoint()** command.

You can add this command to any point in your code (within normal syntactic rules, of course). When the program is run, the execution halts at the point where you inserted the breakpoint command.

The command **exit** finishes the execution of the program.

Using Modules

The **Python standard library** is a collection of standardised functions and objects, which can be used to expand the expressive power of Python in many ways.

The **command import** makes the contents of the given module accessible in the current program.

```
import math

# The square root of the number 5
print(math.sqrt(5))
# the base 2 logarithm of the number 8
```

```
print(math.log(8, 2))
```

Another way to use modules is to select a distinct entity from the module with the **from command**. In case we want to use just the functions sqrt and log from the module math, we can do the following:

```
from math import sqrt, log

print(sqrt(5))
print(log(5,2))
```

The Python documentation has extensive resources on each module in the Python standard library.

We can also have a look at the contents of the module with the **function dir**:

```
import math

print(dir(math))
```

Randomness (random module)

The function **randint(a, b)** returns a random integer value between a and b, inclusive. For example, the following program works like a generic die:

```
from random import randint

print("The result of the throw:", randint(1, 6))
```

The **function shuffle** will shuffle any data structure passed as an argument, in place. For example, the following program shuffles a list of words:

```
from random import shuffle

words = ["atlas", "banana", "carrot"]
shuffle(words)
print(words)
```

The **function choice** returns a randomly picked item from a data structure:

```
from random import choice
```

```
words = ["atlas", "banana", "carrot"]
print(choice(words))
```

The random module contains an even easier way to select lottery numbers: **the sample function**. It returns a random selection of a specified size from a given data structure:

```
from random import sample

number_pool = list(range(1, 41))
weekly_draw = sample(number_pool, 7)
print(weekly_draw)
```

Times and Dates

The **Python datetime module** includes the **function now**, which returns a datetime object containing the current date and time.

You can also define the object yourself:

```
from datetime import datetime

my_time = datetime(1952, 12, 24)
print("Day:", my_time.day)
print("Month:", my_time.month)
print("Year:", my_time.year)
```

The datetime module contains a handy method **strftime** for formatting the string representation of a datetime object.

```
from datetime import datetime

my_time = datetime.now()
print(my_time.strftime("%d.%m.%Y"))
print(my_time.strftime("%d/%m/%Y %H:%M"))
```

Notation	Significance
`%d`	day (01–31)
`%m`	month (01–12)
`%Y`	year in 4 digit format
`%H`	hours in 24 hour format
`%M`	minutes (00–59)
`%S`	seconds (00–59)

Data Processing

Esiste un modulo per leggere più comodamente i file CSV.

```
import csv

with open("test.csv") as my_file:
    for line in csv.reader(my_file, delimiter=";"):
        print(line)
```

The above code reads all lines in the CSV file test.csv, separates the contents of each line into a list using the delimiter ;, and prints each list.

The standard library has a module for working with JSON files: json. The function loads takes any argument passed in a JSON format and transforms it into a Python data structure. So, processing the courses.json file with the code below

```
import json

with open("courses.json") as my_file:
    data = my_file.read()

courses = json.loads(data)
print(courses)
```

```
[{'name': 'Introduction to Programming', 'abbreviation': 'ItP', 'periods': [1, 3]}, {'name': 'Advanced Course in Programming', 'abbreviation': 'ACiP', 'periods': [2, 4]}, {'name': 'Database Application', 'abbreviation': 'DbApp', 'periods': [1, 2, 3, 4]}]
```

The Python standard library also contains modules for dealing with online content, and one useful function is **urllib.request.urlopen**. You are encouraged to have a look at the entire module, but the following example should be enough for you to get to grips with the function. It can be used to retrieve content from the internet, so it can be processed in your programs.

```
import urllib.request

my_request = urllib.request.urlopen("https://helsinki.fi")
print(my_request.read())
```

You may remember from the previous part that you are not allowed to have an empty block in a Python program. If you need to have a block of code which does nothing, for example when testing some other functionality, **the pass command** will let you do this.

In Python, loops can have **else blocks**, too. This section of code is executed if the loop finishes normally.

For example, in the following example we are looking through a list of numbers. If there is an even number on the list, the program prints out a message and the loop is broken. If there are no even numbers, the loop finishes normally, but a different message is then printed out.

A Python function can have a default parameter value. It is used whenever no argument is passed to the function. See the following example:

```
def say_hello(name="Emily"):
    print("Hi there,", name)
```

You can also define a function with a variable number of parameters, by adding a star before the parameter name. All the remaining arguments passed to the function are contained in a tuple, and can be accessed through the named parameter.

```
def testing(*my_args):
    print("You passed", len(my_args), "arguments")
    print("The sum of the arguments is", sum(my_args))

testing(1, 2, 3, 4, 5)
```