Recap and language details for Python exercises

1 - Sequences

The importance of doing exercises

Doing exercises is fundamental

- Making exercises, making errors, trying and trying is fundamental
- Keeping the pace is fundamental in learning programming
- Try to catch up during lectures, with all the exercises provided on Virtuale

Suggested workflow

Today’s exercises are “Exercises 01”

Our advice is to:

- Try to solve the exercise on Thonny (or your preferred IDE)
- Define the function with his body
- Come up with as many tests and prints as you like
- Try the function on Thonny
- Paste **only function definition with its body** on Virtuale and check if it passes the automatic tests

Limits

The automatic test system has limits. They are limits that only apply within it, they are **not absolute Python rules**.

- The **name** of the function and **order** of the parameters must correspond exactly to the ones indicated in the problem text.
- You cannot **ever** use **print** in the automatic test system (unless in exceptional cases, where will be specified).
- Using it would distort the results as incorrect.
- You should only enter the **definition** of the required function (and possibly other useful functions).
- The test prints have already been inserted by us and are invoked during your attempts

Test

- The system tests your function on test cases decided by us. We provided a limited set of test cases. It is up to you to
  - Not hard-code the solution as a series of if, just to pass the tests
  - Come up with other tests for different / edge cases
- During the exam, most of the test cases will be hidden
- You can check and modify the code *as many times as you like* (also during the exam)
- You will see a list of tests with the expected result, the actual result of your program and the overall score assigned
- In the results you will see these colors:
  - red : indicates that all tests are wrong
  - yellow : indicates that some tests (visible and/or hidden) are wrong
  - green : all tests, visible and hidden, have been passed successfully

Scores and ratings

- To each test is assigned a score
- To each exercise is assigned a overall score which is given by the sum of the scores of each test, expressed in a fraction of denominator 10.
- The sum of the scores of all the exercises makes 10
- The "pass" grade is 5/10
- **HOWEVER, all the exercises provided during the lectures** have to be considered **training** evaluation only, **which in no way affects the exam grade**
- It is useful for you to self-assess your skills.
- Moreover, after you submit an *attempt* (i.e. the solution of all the exercises in a quiz), you will have the opportunity to take as many other attempts as you like.
- If you "Review an attempt", you will see a proposed solution for each exercise.

Attention

Attention. There are theoretical, stylistic and efficiency aspects that the automatic system is not able to evaluate.

Review of some topics

Data types

Each data type consists of a set of values and a set of operators handling such values.

Possible types of data:

- integers (int)
- rational numbers (float)
- complex numbers (complex)
- boolean values (bool)
- strings (str)
- ...

Operators for integer numbers, rational numbers and complex numbers

Representations for numerical data types:

- integer: M=3
- rational (float): M=3.0
- complex: M=(3+1j)

Operators:

- M+M → sum (int, float, complex)
- M*M → product (int, float, complex)
- M/M → division (result of type float or complex)
- M//M → quotient (division quotient, result of type int)
- M%M → remainder (division remainder, only for int data type)
- M**M → exponentiation (int, float, complex)

Operations on strings

Representation for strings:

- `M = "Test"; N="home"`

Operators:

- M+N → concatenates strings M and N (ex. `Testhome`)
- M*3 → concatenates 3 times the string in M (ex. `TestTestTest`)
- len(M) → returns the length of string M (ex. `4`)
- M[0], $\cdots$, M[len(M)-1] → returns each single character of string M (ex. M[0] → `T`)

Structure of a function

```
def funcName(parameters):
    """this comment explains the
    function's implementation"""
    instructions
    ...
    instructions #brief comment
    return results
```

Function definition

- A function has got a name
- A function might have parameters (parentheses must be used anyway!)
- A colon (:) is used after a function's definition.
- A function has a content (its *body*).
    - The content of a function must be "indented": spaces must be added at the beginning of each line of the content (the body of a function has to be moved to the right of its definition in order to make the Python interpreter understand that those lines are the function's body). Conventionally **four blank spaces**, or a **tab** are used to indent a function's body.
- A function might return one or more results
- Once a function is defined ...
- ... you might use it as many times you need it.

Comments

- Can be read by a human user but are ignored by the interpreter!
- Comments are used for code documentation
- Conventionally: add a comment just below a function's definition (enclosed within two triplets of double quotes) explaining what the function does.
- Brief comments might be added besides complex instruction (starting with #)

Import module

Modules are pieces of code written to fulfill common tasks (ex. generating random numbers, performing mathematical operations, etc.)

```
from module import ...
```

This command is used to "import" the functions implemented (and saved) in the module. Some commonly used modules are:

- math:
    - contains the implementation of mathematical functions like: `sin(x)`, `cos(x)`, `sqrt(x)`, ...
    - contains the definition of mathematical constants like `e`, `pi`, ... (Neperos number and π)
- random:
    - contains functions to generate random numbers like `randint(a,b)`

Import module

- In order to use the functions defined in some module, within your program:
  - import the module before calling its functions or names, ex: `from math import sin`
  - use the important `sin(90)`
- To know what a module contains check its documentation, ex.: `https://docs.python.org/3/library/math.html` `https://docs.python.org/3/library/random.html`

Input

In Python, to get input from the user, you can use the input function. This function prompts the user for input and returns what he types as a string.

```
>>> input("Enter your name: ")
Enter your name: Angelo
'Angelo'
```

print and input functions are not particularly useful from the Python console, they became very useful when writing programs to be run from files.

Input and assignment

```
<var> = input("input prompt")
```

Input and assignment

```
<var> = input("input prompt")
```

Example:

```
name = input("Enter your name: ")
```

Input and assignment

```
<var> = input("input prompt")
```

Example:

```
name = input("Enter your name: ")
```

- The user sees the message: `Enter your name:`
- The program waits for the users input until he hits the Enter button
- User's input gets stored as a **string** within the name

Input and assignment

```
<var> = input("input prompt")
```

Example:

```
name = input("Enter your name: ")
```

- The user sees the message: `Enter your name:`
- The program waits for the users input until he hits the Enter button
- User's input gets stored as a **string** within the name

```
name = input("Enter your name: ")
print("Hello", name, "!")
```

Input numbers

An input from the keyboard is stored as a string. If we need a numerical input from the user we should convert its type by invoking function **int**() (for conversion to integer numbers), **float**() (for conversion to floating numbers) or **complex**() (for conversion to complex numbers).

```python
age = int(input("Enter your age: "))
```

Assignment

- In Python, `=` **does not literally mean "is equal to"**
- We shall read it from the right to the left; for example in `x = 3+2`
  - the program evaluates (computes) what's on the right of the assignment operator (=)
  - the computed value (5, in this example) is assigned to the name `x`
  - if the name `x` was already assigned a value, the previous assignment (and value) is lost
- Thus: `x = x + 1` has a meaning for assignment!
- While `1 = x` returns an error!
  (1 is not a valid name for a variable identifier)
  the assignment operator (=) **does not satisfy the commutative property**.

Conditional Execution

```
if  condition:
     if internal instructions
    ...
     if internal instructions
 instructions outside the selection scope
```

- The condition (a boolean expression, type **bool** is evaluated
- **If** the boolean expression is evaluated as `True`, the internal instructions of the **if** statement are executed (notice the colon ":" and the indentation).
- If the boolean condition is `False`, the internal instructions of the **if** statement are **not** executed, and the program continues with the instructions outside the selection scope.

Relational Operators

- Here's how relational operators can be written in Python:
  - `x==y` (*x* is equal to *y*)
  - `x!=y` (*x* is not equal to *y*)
  - `x>y` (*x* is greater than *y*)
  - `x<y` (*x* is less than *y*)
  - `x>=y` (*x* is greater than or equal to *y*)
  - `x<=y` (*x* is less than or equal to *y*)

- `x==2`
- `3==x`

  The above expressions are valid syntax (given `x` is defined, both return a boolean value), while...

- `4=x`

  SyntaxError: can't assign to literal

Logical Operators

- (a **and** b) is True if **both** a **and** b are True
- (a **or** b) is True if **either or both of** a **and** b are True
- (**not** a) is True if a is False

In Logic, the operands of the logical operators should be boolean expressions, but Python is not very strict (any non-zero number is interpreted as True):

```
>>> 42 and True
True
>>> 0==False
True
```

Alternative Execution

```
if condition:
     if internal instructions
    ...
     if internal instructions
else:
     else internal instructions
    ...
     else internal instructions
 instructions outside the selection scope
```

- The condition is evaluated.
- **If** the expression is `True` then **only** the if internal instructions are executed.
- **Else** (the condition is `False`), **only** the else internal instructions are executed.
- In both cases, the program execution continues with the instructions outside the selection scope!

Example

```python
def func(param):
    if param=='test':
        print('This is inside the if')
    else:
        print('This is inside the else')
        return
    print('Function main body here!!!')
    return 107
```

With conditional selection a function might follow different branches leading to different return statements: When a return statement is reached the function terminates and the program execution goes back to where the function was called!

Value `None`

`None` is a special value used in Python to mean "no value"

`None` is the value returned by functions with no return statement (or when return statements have no arguments)!

```
def f():
    x = 10
print(f())
```

- `None` is the default value for all expressions that do not return any value.
- The type of this value is `NoneType`: which contains `None` as its only element.

In conditional statements:

- `M == None` → `True` if M has been assigned the `None` value (`f()==None` is True).
- `M != None` → `True` if M has any other value **different** from `None`.

Chained Conditional

```
if condition1:
    if1 internal instr.
elif condition2:
    if2 internal instr.
    ...
elif condition3:
    if3 internal instr.
    ...
else:
    else internal instr.
```

- Use when there are more than two possible cases
- **If** condition1 is `True`, then **only** if1 internal instructions are executed.
- **Else** (if condition1 is `False`), then condition2 is evaluated (first **elif**).
- **If** condition2 is `True`, then **only** if2 internal instructions are executed.
- If condition2 is `False`), the execution moves to the following **elif** .
- and so on...
- If all conditions are `False`, and an **else** statement is given, **only** the else internal instructions are executed.

Sequences

- **Immutable**: cannot be modified after their creation
  - *strings*: sequences of characters
  - *tuples*: sequences of values separated by commas
  - *range*: sequences of integer numbers
- **Mutable**: can be modified after their creation
  - *lists*
  - *dictionaries*
  - ...

Strings

If you want to use text in Python you have to use a string. A string is created by entering text between two single or double quotation marks.

```
>>> "Python is funny"
'Python is funny'
>>> 'Is Python funny?'
'Is Python funny?'
```

When the Python console displays a string it uses single quotes. In any case: the delimiter used for a string doesn't affect how it behaves in any way.

Strings and Quotes

- One, two, or three quotes?
  - You might also use three single quotes: `'''Andrea'''`
  - and freely use single and double quotes within them: `'''He says: "l'm Andrea"'''`
  - Or three double quotes: `"""Andrea"""`
  - You might use single and double quotes within them and create newlines hitting the Return button (works both with three single and three double):

    ```
    """Andrea said: "Hi!"
    She asked: "What's your name?"
    """
    ```

Operations on strings

- We have seen already the sum (+) and multiplication (*) operators applied to strings
- Relational operators can be used too (>, <, ==, ...) for lexicographical (alphabetical) ordering:

  ```
  >>> 'Man' < 'Men'
  True
  ```

- Strings are **immutable** (you might select sub-elements of a string but cannot reassign new values to them):

  ```
  >>> s='test'
  >>> s[0]='F'
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: 'str' object does not support item assignment
  ```

Operations on strings

Selecting characters in a string: `[ ]`

```
Indexes from the start:  0   1   2   3   4   5
Indexes from the end:   -6  -5  -4  -3  -2  -1
                       +---+---+---+---+---+---+
            name  =    | P | y | t | h | o | n |
                       +---+---+---+---+---+---+
```

```
>>> name = 'Python'
>>> name
'Python'
>>> len(name)
6
>>> name[0]
'P'
>>> name[5]
'n'
```

```
>>> name[-1]
'n'
>>> name[-2]
'o'
>>> name[4]
'o'
>>> name[-6] + name[4]
'Po'
```

String operations

Extended *slicing* operator : `[::]`

It has the form `[begin:end:step]`.

If `step` is positive, it goes from `start` included to `end` excluded, taking one element each `step`.

If `step` is negative, we go anyway from `start` included to `end` excluded, taking one element each `step`, but suppose `start >end`. (so going backwards).

```
Indexes from the start:  0   1   2   3   4   5
Indexes from the end:   -6  -5  -4  -3  -2  -1
                       +---+---+---+---+---+---+
            name  =    | P | y | t | h | o | n |
                       +---+---+---+---+---+---+
```

String operations

From the shell:

```
>>> name[0:4:2]
'Pt'
>>> name[-5:-1:2]
'yh'
```

```
>>> name[4:0:-2]
'ot'
>>> nome[-2:-5:-2]
'ot'
```

Range

We use the function **range**`()` to generate sequences of integer numbers:

- **range**`(n)` generates the sequence of integers from 0 to $n-1$
- **range**`(n,m)` generates the sequence of integers from n to $m-1$
- **range**`(n,m,s)` generates the sequence of integers from n to $m-1$ with the given step s

Note that:

- All parameters must be integers
- All parameters can be positive or negative

Example

We use the function **range**() to generate sequences of integer numbers:

- **range**(3) $= < 0, 1, 2 >$
- **range**(2,7) $= < 2, 3, 4, 5, 6 >$
- **range**(0,10,2) $= < 0, 2, 4, 6, 8 >$

- **range**(10,6) $= <>$
- **range**(10,6,-1) $= < 10, 9, 8, 7 >$
- **range**(-7,-1,2) $= < -7, -5, -3 >$

Operator in

It is possible to use **in** and **not in** to know if an element belongs (or not) to a sequence. Check the result of the following (boolean) expressions:

- `'h' in 'Python'`?
- `3 in range(2,9)`?
- `'p' not in 'Python'`?
- `'yth' in 'Python'`? (**in** can match sub-strings in strings)

Within the `string` module a few constants are pre-defined :

```
import string
```

- `string.ascii_letters`
- `string.ascii_lowercase`
- `string.ascii_uppercase`
- `string.digits`
- `string.punctuation`
- `string.whitespace`

- `'3' in string.digits`?

for loops

```
for <var> in <seq>:
    <instructions>
```

- `seq`: a sequence (ex. string, range...)
- `var`: name of the variable that assumes, one by one, in sequence, the values of all the elements in `seq`
- `instructions`: set of instructions to be executed cyclically **for each element** in the sequence

Tuples

- **Ordered sequences** of values (separated by commas):

  ```
  data = 'Anne', 'Smith', 'F', 20
  ```

- Parentheses are optional but commonly used: The IDLE shell uses parentheses when formatting tuples:

  ```
  >>> data
  ('Anne', 'Smith', 'F', 20)
  ```

- Commas are mandatory:
  - `('Michael',)` is a tuple (with just one element)
  - while `('Michael')` is a string!
  - Notice: `()` is an empty tuple!

    ```
    >>> type(())
    <class 'tuple'>
    ```

- The sum (+) operator can be used to concatenate tuples.
- Selection (`[]`) and slicing (`[:]`, `[::]`) operators can be applied to tuples in the usual way:
  - What is `data + ('August',)`
  - What is `data[2]`?

Tuples and assignment

- Values can be packed within a tuple labelled with a single variable:
  `data = ('Anne', 'Smith', 'F', 20)`
- or we can unpack a tuple in a congruous number of variables:
  `(name, surname, sex, age) = data`

Parentheses are optional in both cases! In our example we will have:

```
>>> name
'Anne'
>>> surname
'Smith'
>>> sex
'F'
>>> age
20
>>> data
('Anne', 'Smith', 'F', 20)
```

Ranges and tuples

We know ranges are immutable sequences representing intervals of integer numbers:

- **range**`(n)` is the interval on integer numbers $[0, n[$
- **range**`(a,b)` is the interval on integer numbers $[a, b[$
- **range**`(a, b, s)`, if $s > 0$, is the interval on integer numbers $[a, a + s, a + 2s, ..., a + is]$ with $a + is < b$
- **range**`(a, b, -s)`, if $s > 0$, is the interval on integer numbers $[a, a - s, a - 2s, ..., a - is]$ with $a - is > b$

Ranges and tuples

We know ranges are immutable sequences representing intervals of integer numbers:

- **range**`(n)` is the interval on integer numbers $[0, n[$
- **range**`(a,b)` is the interval on integer numbers $[a, b[$
- **range**`(a, b, s)`, if $s > 0$, is the interval on integer numbers $[a, a+s, a+2s, ..., a+is]$ with $a + is < b$
- **range**`(a, b, -s)`, if $s > 0$, is the interval on integer numbers $[a, a-s, a-2s, ..., a-is]$ with $a - is > b$

To visualize the content of a range we can convert it to a tuple:

```
>>> range(10)
range(0, 10)
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple(range(0,30,5))
(0, 5, 10, 15, 20, 25)
```

```
>>> tuple(range(10,1,-2))
(10, 8, 6, 4, 2)
>>> tuple(range(0))
()
>>> tuple(range(10,0))
()
```

Functions (methods) on strings

In the exercises, you will need to use some functions on strings. In particular

```
>>> str.upper("Ehi!")
"EHI!"
>>> str.lower("Ehi!")
"ehi!"
```

Functions (methods) on strings

In the exercises, you will need to use some functions on strings. In particular

```
>>> str.upper("Ehi!")
"EHI!"
>>> str.lower("Ehi!")
"ehi!"
```

Functions can be found here: `https://docs.python.org/3/library/stdtypes.html#string-methods`

Functions (methods) on strings

In the exercises, you will need to use some functions on strings. In particular

```
>>> str.upper("Ehi!")
"EHI!"
>>> str.lower("Ehi!")
"ehi!"
```

Functions can be found here: `https://docs.python.org/3/library/stdtypes.html#string-methods`

Actually, these are **methods**, you will learn soon to more conveniently write

```
>>> "Ehi!".upper()
"EHI!"
>>> "Ehi!".lower()
"ehi!"
```