Introduction To Python

Part 1/4

Lorenzo Capriotti, Livingstone Ochilo

Outline Of Presentation

- **Programming principles**
- **Essentials of a python program**
  - Python as an object-oriented programming language
  - Using the interactive interpreter
  - Running programs from files
  - Key words
  - Indentation
- **Introduction to basic types, if-statements, functions, loops.**

Programming Principles

- One develops an algorithm: a series of steps that must be followed in order to solve a problem.
- The algorithm is converted into a program, using a programming language e.g. python. This is converted by a computer to machine language.
- Compiled languages – all at once; interpreted languages – line by line.
- Python is an interpreted high-level language.

A Python Program Example

- **A program to display an average of a series of numbers**
  - How will the program get the number? (Keyboard or file)
  - How will it tell the end of the list? (A marker e.g. 0)
  - Should the results be displayed on screen or printed to a file?

  The algorithm may look like this:

  1. Set running total to 0
  2. Set running count to 0.
  3. Repeat these steps:

A Python Program Example [Contd.]

- Read a value
- Check if value is 0 (stop this loop if so)
- Add value to running total.
- Add 1 to running count

4. Compute the average by dividing running total by count
5. Display the average.

– The next step is to test the algorithm using an actual list of numbers.

– How can you modify the program to take care of a list which contains only 0?

Essentials of a Python Program

- **Using the interactive interpreter**
  - Entering *python3* or *python* in the command line should launch the python interpreter.

```
lochilo@ochilo-HP-EliteBook:~$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

  - If you type a number, string or any variable into the interpreter, its value will be echoed to the console

```
>>> "hello"
'hello'
>>> 3
3
```

  - The interpreter is useful for testing code snippets and exploring functions and modules; a program must be written in file to save it.

A Simple Python Programme

This program adds 17 to 20 and prints the results.

```
# This Python program adds 17 and 20 and prints the result.
result = 17 + 20
print("The sum of 17 and 20 is %d." % result)
```

Type it line by line in python and check the output.

Running programs from files

- **Once a written program is saved, it can be run using the** python **command with the file name as a parameter**

```
python myprogram.py
```

- **A python file can be edited using any text editor**
- **Additional packages from the Python Package Index (PyPI) can be installed using a tool such as** pip**.**

Running programs from files

- **A skeleton python program**

```
# Here is the main function.
def my_function():
    print("Hello, World!")

my_function()
```

- **Reserved (key) words**

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

  - These may not be used for any other purpose in a program..

Identifier names

- **Variables, functions and classes used in a program must have unique names (identifiers). Rules for forming an identifier:**
  - It may only contain letters, numbers or the underscore (no spaces)
  - It may not start with a number
  - It may not be a keyword.
- **Meaningful identifiers should:**
  - Be descriptive (avoid unnecessary abbreviations)
  - Be according to a naming convention e.g. classes in CamelCase, variables intended to be constant in CAPITAL_WITH_UNDERSCORES.

Indentation of code

- **Indentation is used in python to delimit blocks, e.g.**

```
# this function definition starts a new block
def add_numbers(a, b):
    # this instruction is inside the block, because it's indented
    c = a + b
    # so is this one
    return c

# this if statement starts a new block
if it_is_tuesday:
    # this is inside the block
    print("It's Tuesday!")
# this is outside the block!
print("Print this no matter what.")
```

  - The end of a line marks the end of instructions, except when the last symbol indicates otherwise.

Indentation of code [Contd.]

Exercise

The following python program is not indented correctly. Rewrite it so that it is correctly indented.

```
def happy_day(day):
if day == "monday":
return ":("
if day != "monday":
return ":D"

print(happy_day("sunday"))
print(happy_day("monday"))
```

Writing Into and Reading Files

Writing to a file using print

```
with open('myfile.txt', 'w') as myfile:
    print("Hello!", file=myfile)
```

Writing to a file using write

```
with open('myfile.txt', 'w') as myfile:
    myfile.write("Hello!")
```

Reading data from a file

```
with open('myfile.txt', 'r') as myfile:
    data = myfile.read()
```

Selection Control Statement: if

- Certain instructions are executed only if certain condition(s) is(are) satisfied. Syntax:

```
if condition:
    if_body
```

- When it reaches an *if* statement, the computer will only execute the body if the condition is met (i.e. if it is true) e.g.

```
if age < 18:
    print("Cannot vote")
```

- If the condition is false, the instructions in the *if* body are skipped.

Python Relational Operators

Operator	Description	Example
==	equal to	if (age == 18)
!=	not equal to	if (score != 10)
>	greater than	if (num_people > 50)
<	less than	if (price < 25)
>=	greater than or equal to	if (total >= 50)
<=	less than or equal to	if (value <= 30)

```
# we can compare the values of strings
if name == "Jane":
    print("Hello, Jane!")

# ... or floats
if size < 10.5:
    print(size)
```

The else Clause

- It enables us to specify alternative instruction(s) if the *if* condition is not met:

```
if condition:
    if_body
else:
    else_body
```

e.g.

```
if x == 0:
    x += 1
else:
    x -= 1
```

Applying the *if*

- **Exercise**

  What is the output of the following code fragment? Explain why.

```
x = 2

if x > 3:
    print("This number")
print("is greater")
print("than 3.")
```

Nested *if* Statements

- An *if* statement is used within the body of another *if* or *else* statement.

```
if weight <= 1000:
    if weight <= 300:
        cost = 5
    else:
        cost = 5 + 2 * round((weight - 300)/100)

    print("Your parcel will cost R%d." % cost)

else:
    print("Maximum weight for small parcel exceeded.")
    print("Use large parcel service instead.")
```

- Bodies of outer if and else clauses are indented
- Bodies of inner if and else clauses are indented one more time.
- Spacing does not alter indentation.

The *elif* Clause

- Long nested if statements can be written more simply using the *elif* clause.

```
if mark >= 80:
    grade = A
else:
    if mark >= 65:
        grade = B
    else:
        if mark >= 50:
            grade = C
        else:
            grade = D
```

```
if mark >= 80:
    grade = A
elif mark >= 65:
    grade = B
elif mark >= 50:
    grade = C
else:
    grade = D
```

Boolean values

- A Boolean value can be either true or false.

```
name = "Jane"

# This is shorthand for checking if name evaluates to True:
if name:
    print("Hello, %s!" % name)

# It means the same thing as this:
if bool(name) == True:
    print("Hello, %s!" % name)

# This won't give us the answer we expect:
if name == True:
    print("Hello, %s!" % name)
```

- The last statement is a string and not a Boolean condition hence gives false output.

Boolean operators: *and*

- A compound expression made up of two sub-expressions and the *and* operator is only true when both sub-expressions are true

```
if mark >= 50 and mark < 65:
    print("Grade B")
```

- More than two sub-expressions can be joined with *and* ; they will be evaluated from left to right

-

```
condition_1 and condition_2 and condition_3 and condition_4
# is the same as
((condition_1 and condition_2) and condition_3) and condition_4
```

Boolean operators: *or, not*

- A compound expression made up of two sub-expressions joined with the or operator is true when at least one of the expressions is true.

```
if age < 0 or age > 120:
    print("Invalid age: %d" % age)
```

The not operator negates an expression; avoid it if possible

```
if name.startswith("A"):
    pass # a statement body can't be empty -- this is an instruction which does nothing.
else:
    print("'%s' doesn't start with A!" % s)

# That's a little clumsy -- let's use "not" instead!
if not name.startswith("A"):
    print("'%s' doesn't start with A!" % s)
```

Exercise

1. For what values of *input* will this program print “True”?

```
if not input > 5:
    print("True")
```

2. For what values of *x* will this program print “True”?

```
if x > 1 or x <= 8:
    print("True")
```

3. Eliminate *not* from each of these boolean expressions:

```
not total <= 2
not count > 40
not (value > 20.0 and total != 100.0)
not (angle > 180 and width == 5)
not (count == 5 and not (value != 10) or count > 50)
not (value > 200 or value < 0 and not total == 0)
```

Lists

- A list is a type of sequence used to store multiple values and access them sequentially by position (index) in the list. Indices are integers, starting from 0.

```
# a list of strings
animals = ['cat', 'dog', 'fish', 'bison']

# a list of integers
numbers = [1, 7, 34, 20, 12]

# an empty list
my_list = []

# a list of variables we defined somewhere else
things = [
    one_variable,
    another_variable,
    third_variable, # this trailing comma is legal in Python
]
```

Lists [Contd.]

- To refer to an element in list, we use the list identifier, followed by index in square brackets.

```
print(animals[0]) # cat
print(numbers[1]) # 7

# This will give us an error, because the list only has four elements
print(animals[6])
```

  Counting from the end

```
print(animals[-1]) # the last element -- bison
print(numbers[-2]) # the second-last element -- 20
```

  We can extract a subset of a list

```
print(animals[1:3]) # ['dog', 'fish']
print(animals[1:-1]) # ['dog', 'fish']
```

- Element at lower bound is included; that at upper bound is excluded.

Lists [Contd.]

- A list is mutable: its elements can be modified, removed, or new ones added.

```
# assign a new value to an existing element
animals[3] = "hamster"

# add a new element to the end of the list
animals.append("squirrel")

# remove an element by its index
del animals[2]
```

- The types of values stored in a list can be mixed.

```
my_list = ['cat', 12, 35.8]
```

Lists [Contd.]

- A list variable can be modified without assigning the variable a completely new value

```
animals = ['cat', 'dog', 'goldfish', 'canary']
pets = animals # now both variables refer to the same list object

animals.append('aardvark')
print(pets) # pets is still the same list as animals

animals = ['rat', 'gerbil', 'hamster'] # now we assign a new list value to animals
print(pets) # pets still refers to the old list

pets = animals[:] # assign a *copy* of animals to pets
animals.append('aardvark')
print(pets) # pets remains unchanged, because it refers to a copy, not the original list
```

Lists [Contd.]

- To check whether a list contains a particular value, we use *in* or *not in* operators.

```
numbers = [34, 67, 12, 29]
my_number = 67

if number in numbers:
    print("%d is in the list!" % number)

my_number = 90
if number not in numbers:
    print("%d is not in the list!" % number)
```

Lists [Contd.]

In-built List functions

```
# the length of a list
len(animals)

# the sum of a list of numbers
sum(numbers)

# are any of these values true?
any([1,0,1,0,1])

# are all of these values true?
all([1,0,1,0,1])
```

Other useful methods

```
numbers = [1, 2, 3, 4, 5]

# we already saw how to add an element to the end
numbers.append(5)

# count how many times a value appears in the list
numbers.count(5)
```

List Object Methods

```
# append several values at once to the end
numbers.extend([56, 2, 12])

# find the index of a value
numbers.index(3)
# if the value appears more than once, we will get the index of the first one
numbers.index(2)
# if the value is not in the list, we will get a ValueError!
numbers.index(42)

# insert a value at a particular index
numbers.insert(0, 45) # insert 45 at the beginning of the list

# remove an element by its index and assign it to a variable
my_number = numbers.pop(0)

# remove an element by its value
numbers.remove(12)
# if the value appears more than once, only the first one will be removed
numbers.remove(5)
```

Exercise

1. Create a list *a* which contains the first three odd positive integers and *a* list *b* which contains the first three even positive integers.
2. Create a new list *c* which combines the numbers from both lists (order not important).
3. Create a new list *d* which is a sorted copy of *c*, leaving *c* unchanged.
4. Reverse *d* in-place.
5. Set the fourth element of *c* to 42
6. Append 10 to the end of *d.*
7. Print the length of *d.*
8. Print the last element of d without using its length.


To be continued...