Intro to programming 2

Henri Vandendriessche
henri.vandendriessche@ens.fr

2023-09-26

Terminal cheat sheet reminder

- Bash commands to navigate directories
  - Print Working Directory. Print the path of the current directory

```
pwd
```

  - List all files of the current directory

```
ls folder
```

  - Moving into folder1 and subfolder2 at once.

```
cd folder1/subfolder2
```

  - Moving out of a directory

```
cd ..
```

  - Going back to the root directory

```
cd ~
```

- "**Tab**" to use the auto-completion
- "**Upper arrow**" to see last commands
- **Ctrl + C** to stop a program execution
- Many more bash commands to use. . .

So far

- Python
- Variables
- Data types:
  - integer
  - float
  - string
  - boolean
- If and For loops:
  - syntax use the right keywords **if**, **elif**, **else**, **for**, **in**
  - don't forget the **:**
  - and the indentation

Reading advice

To complete what we're going to see today.

- https://automatetheboringstuff.com/2e/chapter4/
- https://automatetheboringstuff.com/2e/chapter5/

Today

- Constant and Variable
- While loop
- Other python data types for collections of data type
    - list
    - set
    - tuple
    - dictionary
- Exercises

Constant vs Variable

- So far we have seen how to declare, instantiate and modify variables

Constant vs Variable

- So far we have seen how to declare, instantiate and modify variables
- However, there are “variables” that you don’t want to modify, and traditionally in programming, these variables are called constants.

Constant vs Variable

- So far we have seen how to declare, instantiate and modify variables
- However, there are “variables” that you don’t want to modify, and traditionally in programming, these variables are called constants.
- Unfortunately, there is no native way of declaring a constant in Python. However, there is an unwritten convention that you should use only uppercase letters.

```
CONST_PI = 3.1415
```

Constant vs Variable

- So far we have seen how to declare, instantiate and modify variables
- However, there are “variables” that you don’t want to modify, and traditionally in programming, these variables are called constants.
- Unfortunately, there is no native way of declaring a constant in Python. However, there is an unwritten convention that you should use only uppercase letters.

```
CONST_PI = 3.1415
```

- Python includes some constants in its core library.

```
import math

math.pi
```

```
## 3.141592653589793
```

While loop 1/3

- **While** loops will keep looping until an ending condition is met

While loop 1/3

- **While** loops will keep looping until an ending condition is met
- Unlike the **for** loop, which incorporates a specified number of executions, the **while** loop is best suited for situations with an unknown or very large number of loop iterations.

While loop 1/3

- **While** loops will keep looping until an ending condition is met
- Unlike the **for** loop, which incorporates a specified number of executions, the **while** loop is best suited for situations with an unknown or very large number of loop iterations.
- Similar to **if** and **for**, the **while** loop has a syntax with ':' and requires indentation for the following lines to be included in the process.

While loop 1/3

- **While** loops will keep looping until an ending condition is met
- Unlike the **for** loop, which incorporates a specified number of executions, the **while** loop is best suited for situations with an unknown or very large number of loop iterations.
- Similar to **if** and **for**, the **while** loop has a syntax with ':' and requires indentation for the following lines to be included in the process.
- The two key features of a **while** loop are:

While loop 1/3

- **While** loops will keep looping until an ending condition is met
- Unlike the **for** loop, which incorporates a specified number of executions, the **while** loop is best suited for situations with an unknown or very large number of loop iterations.
- Similar to **if** and **for**, the **while** loop has a syntax with ':' and requires indentation for the following lines to be included in the process.
- The two key features of a **while** loop are:
    - the output condition

While loop 1/3

- **While** loops will keep looping until an ending condition is met
- Unlike the **for** loop, which incorporates a specified number of executions, the **while** loop is best suited for situations with an unknown or very large number of loop iterations.
- Similar to **if** and **for**, the **while** loop has a syntax with ':' and requires indentation for the following lines to be included in the process.
- The two key features of a **while** loop are:
  - the output condition
  - the increment statement

While loop 2/3

- Example :

```
i = 1
while i < 4: #  output condition
  print(i)
  i += 1      # increment statement
```

```
## 1
## 2
## 3
```

While loop 2/3

- Example :

```
i = 1
while i < 4: #  output condition
  print(i)
  i += 1      # increment statement
```

```
## 1
## 2
## 3
```

- Which is technically the same as

```
for i in range(1,4):
  print(i)
```

```
## 1
## 2
## 3
```

While loop 3/3

- The **While** loop will test if the output condition is True, and if not, it will execute the code and then execute the increment statement.

While loop 3/3

- The **While** loop will test if the output condition is True, and if not, it will execute the code and then execute the increment statement.
- If either of these two conditions is not correctly specified, you may encounter an error or an infinite loop. . . .

While loop 3/3

- The **While** loop will test if the output condition is True, and if not, it will execute the code and then execute the increment statement.
- If either of these two conditions is not correctly specified, you may encounter an error or an infinite loop. . . .
- Example 1

```
i = 1
while i < 6: #  output condition
  print(i)
```

While loop 3/3

- The **While** loop will test if the output condition is True, and if not, it will execute the code and then execute the increment statement.
- If either of these two conditions is not correctly specified, you may encounter an error or an infinite loop. . . .
- Example 1

```
i = 1
while i < 6: #  output condition
  print(i)
```

- Example 2

```
i = 1
while i != 6:
  print(i)
  i += 2
```

Warning on while loop

- A **While** loop cannot directly iterate over the elements of a sequence like the **for** loop

```
list1 = [1,2,3,0]
while x in list1:
  print(x)
```

NameError: name ‘x’ is not defined

Breaking a loop 1/2

- You can break out of a loop using the **break** statement. This is useful, for example, when the remaining iterations of the loop are unnecessary.

```
# Checking if a number is primitive
N = 72239
for i in range(2, 300):
  if N % i == 0:
    print(i)
    break
```

```
## 29
```

- Other example

```
# Checking a password
passwd = 'sesame'

while True:
  code = input('Password? ')
  if code == passwd:
    break
  else:
    print('invalid password')

print("You are in!")
```

Breaking a loop 2/2

- The keyword **continue** also very useful for skipping the current iteration.is also very useful to pass the current iteration

```
for i in range(0,5):
  if i == 3:
    continue
  else:
    print(i)
```

```
## 0
## 1
## 2
## 4
```

Lists 1/3

- A list is a collection of related objects

Lists 1/3

- A list is a collection of related objects
- It is declared using square brackets [] with commas separating the objects.

Lists 1/3

- A list is a collection of related objects
- It is declared using square brackets [] with commas separating the objects.
- Example 1

```
Years_France_won_worldcup = [1998, 2018]
print(Years_France_won_worldcup)
```

```
## [1998, 2018]
```

Lists 1/3

- A list is a collection of related objects
- It is declared using square brackets [] with commas separating the objects.
- Example 1

```
Years_France_won_worldcup = [1998, 2018]
print(Years_France_won_worldcup)
```

```
## [1998, 2018]
```

- Example 2

```
dog_breeds = ["golden", "corgi", "Bulldog", "Husky", "Beagle"]
dog_breeds2 = ["golden" "corgi" "Bulldog" "Husky" "Beagle"]
print(dog_breeds)
```

```
## ['golden', 'corgi', 'Bulldog', 'Husky', 'Beagle']
```

```
print(dog_breeds2)
```

```
## ['goldencorgiBulldogHuskyBeagle']
```

Lists 1/3

- A list is a collection of related objects
- It is declared using square brackets [] with commas separating the objects.
- Example 1

```
Years_France_won_worldcup = [1998, 2018]
print(Years_France_won_worldcup)
```

```
## [1998, 2018]
```

- Example 2

```
dog_breeds = ["golden", "corgi", "Bulldog", "Husky", "Beagle"]
dog_breeds2 = ["golden" "corgi" "Bulldog" "Husky" "Beagle"]
print(dog_breeds)
```

```
## ['golden', 'corgi', 'Bulldog', 'Husky', 'Beagle']
```

```
print(dog_breeds2)
```

```
## ['goldencorgiBulldogHuskyBeagle']
```

- Example 3

```
random_data_type_collection = [ 1, True, "Cats", 3.14]
print(random_data_type_collection)
```

```
## [1, True, 'Cats', 3.14]
```

Lists 2/3

- You can access elements in a list through their index, which is the same as accessing characters in a string.

```
prog_language = ["python", "R", "C", "java", "Go", "Rust"]
print(prog_language[0])
```

```
## python
```

```
print(prog_language[-1])
```

```
## Rust
```

```
programming_language = "python"
print(programming_language[0])
```

```
## p
```

```
print(type(programming_language))
```

```
## <class 'str'>
```

Lists 3/3

- Lists, like any other variables, can be used with functions.

Lists 3/3

- Lists, like any other variables, can be used with functions.
  - append()

Lists 3/3

- Lists, like any other variables, can be used with functions.
    - append()
    - remove()

Lists 3/3

- Lists, like any other variables, can be used with functions.
    - append()
    - remove()
    - pop()

Lists 3/3

- Lists, like any other variables, can be used with functions.
    - append()
    - remove()
    - pop()
    - sort()

Lists 3/3

- Lists, like any other variables, can be used with functions.
    - append()
    - remove()
    - pop()
    - sort()
    - len()

Lists 3/3

- Lists, like any other variables, can be used with functions.
  - append()
  - remove()
  - pop()
  - sort()
  - len()

- Example

```
prog_language = ["python", "R", "C", "java", "Go", "Rust"]
prog_language.append("html")
prog_language.append("PHP")
print(prog_language)
```

```
## ['python', 'R', 'C', 'java', 'Go', 'Rust', 'html', 'PHP']
```

```
prog_language.remove("html")

len(prog_language)
```

```
## 7
```

```
prog_language.sort()
print(prog_language)
```

```
## ['C', 'Go', 'PHP', 'R', 'Rust', 'java', 'python']
```

Tuples 1/3

- **Tuples** are very similar to Lists and is used for data collection

Tuples 1/3

- **Tuples** are very similar to Lists and is used for data collection
- They are declared with parentheses **()** instead of square brackets and with commas **,** to separate the objects.

Tuples 1/3

- **Tuples** are very similar to Lists and is used for data collection
- They are declared with parentheses **()** instead of square brackets and with commas **,** to separate the objects.
- Example

```
date_covid_shots = ("21-04-15", "21-05-18", "21-09-20")

print(type(date_covid_shots))
```

```
## <class 'tuple'>
```

```
print(date_covid_shots[1])  # Accessible as list with index with []
```

```
## 21-05-18
```

```
print(len(date_covid_shots))
```

```
## 3
```

Tuples 2/3

- In contrast to lists, they are immutable and can't be modified.

```
date_covid_shots = ("21-04-15", "21-05-18", "21-09-20")
date_covid_shots.append("21-09-27")
```

```
## 'tuple' object has no attribute 'append'
```

Tuples 2/3

- In contrast to lists, they are immutable and can't be modified.

```
date_covid_shots = ("21-04-15", "21-05-18", "21-09-20")
date_covid_shots.append("21-09-27")
```

```
## 'tuple' object has no attribute 'append'
```

- You can't change the order of items or modify the value of an item.

Tuples 2/3

- In contrast to lists, they are immutable and can't be modified.

```
date_covid_shots = ("21-04-15", "21-05-18", "21-09-20")
date_covid_shots.append("21-09-27")
```

```
## 'tuple' object has no attribute 'append'
```

- You can't change the order of items or modify the value of an item.
- Tuples are best suited when you need ordered lists that would never change

Tuples 2/3

- In contrast to lists, they are immutable and can't be modified.

```
date_covid_shots = ("21-04-15", "21-05-18", "21-09-20")
date_covid_shots.append("21-09-27")
```

```
## 'tuple' object has no attribute 'append'
```

- You can't change the order of items or modify the value of an item.
- Tuples are best suited when you need ordered lists that would never change
    - For example, if you want to represent a calendar, days and years can be coded as tuples since they won't change but are ordered.

Tuples 3/3

- Note that you can combine lists and tuples

```
Cocktails = [("Cosmo","5€"),("Daiquiri","7€"),("B52","6€")]
Cocktails.append(("Mojito","7€"))

print(Cocktails)
```

```
## [('Cosmo', '5€'), ('Daiquiri', '7€'), ('B52', '6€'), ('Mojito', '7€')]
```

Tuples 3/3

- Note that you can combine lists and tuples

```
Cocktails = [("Cosmo","5€"),("Daiquiri","7€"),("B52","6€")]
Cocktails.append(("Mojito","7€"))

print(Cocktails)
```

```
## [('Cosmo', '5€'), ('Daiquiri', '7€'), ('B52', '6€'), ('Mojito', '7€')]
```

- NB: You can also declare a tuple using the **tuple()** constructor.

```
date_covid_shots = tuple(["21-04-15", "21-05-18", "21-09-20"])
# in this line you transform a list into a tuple
print(type(date_covid_shots))
```

```
## <class 'tuple'>
```