

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...

- 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** 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
```


- 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.

- 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.

- 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'>
```

Sets 1/2

- **Sets** are very similar to **lists** but differ in that they are unordered, unindexed, and do not allow duplicate values. However, they are mutable.

Sets 1/2

- **Sets** are very similar to **lists** but differ in that they are unordered, unindexed, and do not allow duplicate values. However, they are mutable.
- Sets are declared using curly braces `{}` and items again are separated with `,`.

Sets 1/2

- **Sets** are very similar to **lists** but differ in that they are unordered, unindexed, and do not allow duplicate values. However, they are mutable.
- Sets are declared using curly braces `{}` and items again are separated with `,`.
- Example

```
fruit_I_like = {"apple", "pineapple", "peach"}  
print(type(fruit_I_like))
```

```
## <class 'set'>
```

```
print(fruit_I_like)
```

```
## {'pineapple', 'apple', 'peach'}
```

```
fruit_I_like.add("strawberry")
```

```
"strawberry" in fruit_I_like # Check if a fruit is in my set
```

```
## True
```

```
fruit_I_like.remove("apple")  
print(fruit_I_like)
```

```
## {'pineapple', 'strawberry', 'peach'}
```

- You cannot add or remove items in a **set**, or access items with an index. Attempting to do so will result in an error.

```
fruit_I_like[0]
```

```
## 'set' object is not subscriptable
```

```
fruit_I_like.append("banana")
```

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

- You cannot add or remove items in a **set**, or access items with an index. Attempting to do so will result in an error.

```
fruit_I_like[0]
```

```
## 'set' object is not subscriptable
```

```
fruit_I_like.append("banana")
```

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

- The **append()** method doesn't work with **sets** because it implies an order to add an item at the end, which sets do not have.

- You cannot add or remove items in a **set**, or access items with an index. Attempting to do so will result in an error.

```
fruit_I_like[0]
```

```
## 'set' object is not subscriptable
```

```
fruit_I_like.append("banana")
```

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

- The **append()** method doesn't work with **sets** because it implies an order to add an item at the end, which sets do not have.
- NB: you can also declare a set using a **set()** constructor (as for the tuple)

```
date_covid_shots = set(["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 'set'>
```

- **Dictionaries** are data structure that uses data in key-value pairs.

- **Dictionaries** are data structure that uses data in key-value pairs.
- Each item of a dictionary is a key-value pair.

- **Dictionaries** are data structure that uses data in key-value pairs.
- Each item of a dictionary is a key-value pair.
- Each key in a dictionary must be unique.

- **Dictionaries** are data structure that uses data in key-value pairs.
- Each item of a dictionary is a key-value pair.
- Each key in a dictionary must be unique.
- Dictionaries are declared in a very specific way.

- **Dictionaries** are data structure that uses data in key-value pairs.
- Each item of a dictionary is a key-value pair.
- Each key in a dictionary must be unique.
- Dictionaries are declared in a very specific way.
 - `my_dictionary = { "key1" : value1, "key2": value2 ... }`

Dictionaries 2/5

- Example

```
PCBS = {  
    "Name" : "PCBS",  
    "Teacher" : "Christophe Pallier",  
    "Teacher assistant" : "Henri",  
    "Course" : ["1", "2"],  
    "Day": "Tuesday",  
    "Duration" : 3,  
    "Mandatory" : False}
```

```
print(PCBS["Name"])
```

```
## PCBS
```

```
print(PCBS["Teacher"])
```

```
## Christophe Pallier
```

```
print(PCBS["Teacher assistant"])
```

```
## Henri
```

```
print(PCBS["Course"])
```

```
## ['1', '2']
```

```
print(PCBS["Day"])
```

```
## Tuesday
```

Dictionaries 3/5 Create a Dictionary

- But you have several methods to create a dictionary:

Dictionaries 3/5 Create a Dictionary

- But you have several methods to create a dictionary:
- Example1

```
PCBS = {}  
PCBS["Name"] = "PCBS"  
PCBS["Teacher"] = "Christophe Pallier"  
PCBS["Teacher assistant"] = "Henri"  
PCBS["Course"] = ["1", "2"]  
PCBS["Day"] = "Tuesday"  
PCBS["Duration"] = 3  
PCBS["Mandatory"] = False
```

```
print(PCBS)
```

```
## {'Name': 'PCBS', 'Teacher': 'Christophe Pallier', 'Teacher assistant': 'Henri'}
```

```
print(type(PCBS))
```

```
## <class 'dict'>
```

Dictionaries 4/5 Create a Dictionary

- Which is exactly the same as

```
PCBS = dict()
PCBS["Name"] = "PCBS"
PCBS["Teacher"] = "Christophe Pallier"
PCBS["Teacher assistant"] = "Henri"
PCBS["Course"] = ["1", "2"]
PCBS["Day"] = "Tuesday"
PCBS["Duration"] = 3
PCBS["Mandatory"] = False
print(PCBS)
```

```
## {'Name': 'PCBS', 'Teacher': 'Christophe Pallier', 'Teacher assistant': 'Henri'}
```

Dictionaries 4/5 Create a Dictionary

- Which is exactly the same as

```
PCBS = dict()
PCBS["Name"] = "PCBS"
PCBS["Teacher"] = "Christophe Pallier"
PCBS["Teacher assistant"] = "Henri"
PCBS["Course"] = ["1", "2"]
PCBS["Day"] = "Tuesday"
PCBS["Duration"] = 3
PCBS["Mandatory"] = False
print(PCBS)
```

```
## {'Name': 'PCBS', 'Teacher': 'Christophe Pallier', 'Teacher assistant': 'Henri'}
```

- But with a constructor dict()

Dictionaries 5/5 Common operation

- Access to a a key-value pair

Dictionaries 5/5 Common operation

- Access to a a key-value pair
- Add a key-value pair

Dictionaries 5/5 Common operation

- Access to a a key-value pair
- Add a key-value pair
- Delete a key-value pair

Dictionaries 5/5 Common operation

- Access to a a key-value pair
- Add a key-value pair
- Delete a key-value pair
- Check for specific key existence

Dictionaries 5/5 Common operation

- Access to a key-value pair
- Add a key-value pair
- Delete a key-value pair
- Check for specific key existence
- Example

```
PCBS = { "Name" : "PCBS", "Teacher" : "Christophe Pallier",  
        "Teacher assistant" : "Henri",  
        "Day": "Tuesday", "Duration" : 3, "Mandatory" : False}  
PCBS["Course"] = ["1", "2"]  
PCBS['Course'].append("3")  
PCBS['Day']
```

```
## 'Tuesday'
```


```
PCBS["starting time"] = "14h00"  
PCBS.pop("Teacher assistant")
```

```
## 'Henri'
```

```
print(PCBS)
```

```
## {'Name': 'PCBS', 'Teacher': 'Christophe Pallier', 'Day': 'Tuesday', 'Duration': 3, 'Course': ['1', '2', '3'], 'starting time': '14h00'}
```

Summary on Python collections (~ Arrays)

	List	tuple	Set	Dictionary
Mutable	✓	✗	✓	✓
Ordered	✓	✓	✗	✓
Indexing	✓	✓	✗	✓
Duplicate elements	✓	✓	✗	 values can be duplicated Keys can't
Can be created using	list()	tuple()	set()	dict()

Exercises

- Exercise 1: Lists: `list1 = [1,2,3,4,1]`
 - Given list1 print their sum with for and while loops
 - Given list1 print their product for and while loops
 - Given list1 print the sum of their squares for and while loops
 - Given list1 print the largest number for and while loops
 - Given list1 print the second largest for and while loops
- Exercise 2: Tuples
 - Given a list `l=[1, 2, 3, 6, 7, 4, 5]`, transform it into a tuple
 - Return the min and max of each tuples: `tuple = [(1,3,2), (6,4,5), (8,7,9)]`
 - Given a list of tuples, return tuples that have all positive elements. `test_tuples = [(1,2,3), (4,5,6), (7,8,9), (-1,2,3)]`
- Exercise 3 : Sets
 - Order the tuples l from Exercise 2 and transform it into a Set
 - Given a set `Set1 = { 1,2,3,3,5,6,7}` remove the 4th items
 - Given two sets a, b. Print True if they have items in common or False if not. `a = {"apple", "pineapple", "peach", "pears", "lemon", "lychee"}` `b = {"banana", "mango", "lychee", "kiwi", "apple", "orange"}`
- Exercise 4: Given a list of words, count the number of times each word appears in the list (using dictionary)
 - `animaList=["dog", "horse", "cat", "fish", "cat", "fox", "tiger", "tiger", "flamingo", "cat"]`