01 Python

Introduction to Python

Author: Denis Rinfret

About Python

> Python is powerful… and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open.

> These are some of the reasons people who use Python would rather not use anything else.

https://www.python.org/about/

Getting Started

Python can be easy to pick up whether you're a first time programmer or you're experienced with other languages.

- Beginner's Guide, Programmers
- Beginner's Guide, Non-Programmers
- Beginner's Guide, Download & Installation
- Code sample and snippets for Beginners

python.org

Friendly & Easy to Learn

- The community hosts conferences and meetups, collaborates on code, and much more.
- Python's documentation will help you along the way, and the mailing lists will keep you in touch.

Open-source

- Python is developed under an OSI-approved open source license, making it freely usable and distributable, even for commercial use.

- Python’s license is administered by the Python Software Foundation.

python.org

Applications

- The Python Package Index (PyPI) hosts thousands of third-party modules for Python.
- Both Python’s standard library and the community-contributed modules allow for endless possibilities:
    - Web and Internet Development
    - Database Access
    - Desktop GUIs
    - Scientific & Numeric
    - Education
    - Network Programming
    - Software & Game Development

python.org

Python Version

- Any python version 3.10 or higher will work for this course
- It should already be installed if you use any Linux distribution or MacOS
- On Windows, download the latest stable version from here: https://www.python.org/downloads/

PyCharm IDE

- You should use the PyCharm IDE, professional version, available from https://www.jetbrains.com/pycharm/download/, as explained earlier
- *Reminder*:
    - Download PyCharm from JetBrains.com
    - Choose the *Professional* version.
    - Apply for a *Student* license here
    - Use your *university email address* to apply
    - Or you can use your GitHub account if you already have a GitHub Student Developer Pack

- You can use another IDE or text editor, but it might be harder to follow the presentations and it might be harder to get help from the professor and from the tutors
- If you are already using another JetBtains IDE, such as IntelliJ IDEA, you can install the Python plugin, which will enable most of the PyCharm functionality

First Python Program

```
print("Hello, Python!")
```

```
Hello, Python!
```

- Python is a scripting language
- it is interpreted and dynamically typed
- proper indentation is a must
    - no curly brackets `{}` for blocks of code
    - no semi-colon at the end of a statement or line
- Python supports *procedural* and *Object-Oriented* programming styles, as well as many *functional* programming constructs

Strings

String literals in Python can be defined with a pair of double quotes, single quotes, triple double quotes or triple single quotes.

```
str1 = "a string with double quotes"
str2 = 'a string with single quotes'
str3 = """a string with triple double quotes"""
str4 = '''a string with triple single quotes'''
```

Triple double and triple single quoted strings can be multi-line strings.

```
str3 = """a string with
triple double quotes"""
str4 = '''a string
with triple
single quotes'''
```

Variables

- Variables don’t need to be declared with a type
- Simply assign to a variable to create it
- If already created, the assignment will replace the old value by a new one

```
x = 5  # creates a new variable and assigns 5 to it
x = 10  # replaces the value 5 in x by the value 10
```

Variable Types

- Variables don’t have types in Python
- All variables actually hold references to some object
- The objects referred to by the variables have a type, but the variables themselves don’t have a type

```
name = "Denis"
name = 4
```

- The code above is legal, although assigning the integer `4` to a variable `name` doesn’t make much sense

Integers and Floating-Point Numbers

- Be careful, Python doesn’t handle division between integers the same way as most other programming languages

```
print(10 / 3)
```

```
3.3333333333333335
```

- The single slash `/` is actually a *float* division in Python, even though both operands are integers
- Use the double slash `//` to get an integer division
- Use can use the modulo operator `%` to get the remainder

```
print(10 // 3, 10 % 3)
```

```
3 1
```

- Integers in Python are *infinite precision* integers
- They will not overflow like in most other progamming languages

```
# the double star ** is the power (or exponent) operator
x = 2 ** 100
print(x)
# the hex() function converts an integer into an hexadecimal string
print(hex(x))
```

```
1267650600228229401496703205376
0x10000000000000000000000000
```

Lists

- There are no arrays in the core Python programming language
- Instead, we use lists to get similar functionality
- Arrays can be faster than lists in some cases, but lists are more convenient in many (even most) cases

```
scores = [88, 52, 69, 100, 89, 78, 95, 75]
print(scores)
print(scores[2])
scores[0] = 90
print(scores)
```

```
[88, 52, 69, 100, 89, 78, 95, 75]
69
[90, 52, 69, 100, 89, 78, 95, 75]
```

Out of bounds indexes

```
print(scores[100])
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)
```

```
Cell In [10], line 1
----> 1 print(scores[100])
```

```
IndexError: list index out of range
```

```
# negative indexes start from the end
print(scores[-1])
```

```
75
```

```
print(scores[-100])
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

Cell In [12], line 1
----> 1 print(scores[-100])


IndexError: list index out of range
```

Sublists

```
print(scores)
# the first index is inclusive
# the second index is exclusive
print(scores[2:6])
```

```
[90, 52, 69, 100, 89, 78, 95, 75]
[69, 100, 89, 78]
```

```
print(scores[:6])  # missing first index defaults to 0
```

```
[90, 52, 69, 100, 89, 78]
```

```
print(scores[2:])  # missing second index defaults to len(scores)
```

```
[69, 100, 89, 78, 95, 75]
```

```
print(scores[:-1])  # can use negative indexes too
```

```
[90, 52, 69, 100, 89, 78, 95]
```

```
print(scores[2:6:2])  # the third index is the step; defaults to 1
```

```
[69, 89]
```

Exercises

1. How do you get a sublist with only the elements with an **even** index value?
2. How do you get a sublist with only the elements with an **odd** index value?

Loops and Conditionals

Counted Loops with Index Variable in a Range of Values

```
for i in range(5):  # 0 (inclusive) to 5 (exclusive)
    print(i)
```

```
0
1
2
3
4
```

- The colon `:` is mandatory, it denotes the start of a block
- Blocks must be indented
- To end a block, simply *unindent* the first statement outside and after the block

```python
for i in range(5):  # 0 (inclusive) to 5 (exclusive)
    print(i)
print("Done!")  # What happens if you indent this statement?
```

```
0
1
2
3
4
Done!
```

```python
for i in range(3, 10):  # 3 (inclusive) to 10 (exclusive)
    print(i)
```

```
3
4
5
6
7
8
9
```

```python
# 3 (inclusive) to 10 (exclusive), steps of 2
for i in range(3, 10, 2):
    print(i)
```

```
3
5
7
9
```

```python
# 10 (inclusive) to 3 (exclusive), steps of -2
for i in range(10, 3, -2):
    print(i)
```

```
10
8
```

```
6
4
```

```
# 10 (inclusive) to 3 (exclusive), steps of 2
# gives an empty range
for i in range(10, 3, 2):
    print(i)
```

Loops on the Elements of a List

```
for i in range(len(scores)):  # go through all elements of a list
    print(i, scores[i])
```

```
0 90
1 52
2 69
3 100
4 89
5 78
6 95
7 75
```

```
for x in scores:  # often called a foreach loop in other languages
    print(x)
```

```
90
52
69
100
89
78
95
75
```

```
# loop on indexes and elements of list
for i, score in enumerate(scores):
    print(i, score)
```

```
0 90
1 52
2 69
3 100
4 89
5 78
6 95
7 75
```

Exercises

1. Write a loop to add up all the numbers from 1 to \(n\) (inclusive). \[\sum_ {i=1}^{n}i\] Start by creating a variable named `n`, equal to some positive integer. Print the total at the end.

2. Write a loop to add up all the numbers in a list. Start by creating a list containing some numbers. Print the total at the end.

3. Write a loop to add up all the numbers with an **even index** in a list. Start by creating a list containing some numbers. Print the total at the end.

Conditionals

```
x = -5
if x < 0:  # what does this if statement do?
    x = -x
print(x)
```

```
5
```

Example: converting scores into letter grades

```
print(scores)
if scores[0] >= 90:  # no parentheses needed around condition
    print("A")
```

```
else:
    print("Not A")
```

```
[90, 52, 69, 100, 89, 78, 95, 75]
A
```

- Be careful with indentation: Python requires proper indentaion to work!
- All statements in the same block must be at the same indentation level
- The `if` and corresponding `else` must be at the same indentation level (i.e. same column)
- If multiple statements are needed in the `if` block or `else` block, then they must all be at the same indentation level

```
if scores[0] >= 90:
    print("A")
elif scores[0] >= 80:
    print("B")
else:
    print("Less than B")
```

```
A
```

- There are no *switch* or *case* statements in Python
- Use a `if ... elif ... elif ... ... else ...` sequence instead
- It's possible to do a kind of *switch* with dictionaries or classes (to be covered later)

```
# loop over all the scores and convert all grades
for score in scores:
    if score >= 90:
        print("A")
    elif score >= 80:
        print("B")
    else:
        print("Less than B")
```

```
A
Less than B
Less than B
A
B
```

```
Less than B
A
Less than B
```

Example: check invalid scores

```python
score = 12
if score < 0 or score > 100:
    print("Invalid score!")
```

Example: check failing scores

```python
if score >= 0 and score < 50:
    print("Not good")
if 0 <= score < 50:
    print("Not good")
```

```
Not good
Not good
```

While Loops

- Similar syntax to `if`

```python
x = 5
while x > 0:
    print(x)
    x -= 1
```

```
5
4
3
2
1
```

- Use `break` to get out of a loop
- Use `continue` to skip the rest of the loop body and continue the loop
- Loops can also have an `else` block, executed just after the last “turn” of the loop

```python
x = 4
while x > 0:
    print(x)
    x -= 1
else:
    print("in the else, x =", x)
```

```
4
3
2
1
in the else, x = 0
```

- Will the `else` be executed on a `break` ?

```python
x = 5
while True:
    if x <= 0:
        break
    print(x)
    x -= 1
else:
    print("in the else, x =", x)
```

```
5
4
3
2
1
```

Dictionaries

- Also called *maps* and *associative arrays* in other programming languages
- Similar to lists, but use *keys* instead of *indexes* to access values stored in the dictionary
- Keys can be of almost any type (they must be hashable)
- Keys are often strings, but not always
- **Major difference** with lists: dictionaries are not **sorted**, meaning that (key, value) pairs are not kept in any specific order

```python
data = {"firstname": "John", "lastname": "Doe", "age": 88}
print(data)
print(data["lastname"])
data["age"] = 66
data["abc"] = 123
del data["age"]
print(data)
```

```
{'firstname': 'John', 'lastname': 'Doe', 'age': 88}
Doe
{'firstname': 'John', 'lastname': 'Doe', 'abc': 123}
```

Looping on Dictionaries

```python
for k in data:  # loop on keys
    print(k)
```

```
firstname
lastname
abc
```

```python
for k in data:  # loop on keys, get associated values
    print(k, data[k])
```

```
firstname John
lastname Doe
abc 123
```

```python
for k, v in data.items():  # loop on dictionary items
    print(k, v)
```

```
firstname John
lastname Doe
abc 123
```

```
for k in sorted(data):  # loop on sorted keys
    print(k, data[k])
```

```
abc 123
firstname John
lastname Doe
```

```
for k, v in sorted(data.items()):  # loop on sorted items
    print(k, v)
```

```
abc 123
firstname John
lastname Doe
```

```
# loop on reverse sorted items
for k, v in reversed(sorted(data.items())):
    print(k, v)
```

```
lastname Doe
firstname John
abc 123
```

Comprehensions

List Comprehensions

- Build a list from another list, by transforming the elements

Example: build a list of cube values

```
cubes = [x ** 3 for x in range(10)]
print(cubes)
cubes = []
for x in range(10):
    cubes.append(x ** 3)
print(cubes)
```

```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

- the square brackets `[]` are used to create a list
- the elements are not listed explicitely: instead, they are computed from another list or sequence
    - in this case, we compute the cube of a number `x**3`
    - and we apply the computation to a sequence of numbers created with the `range` function
    - but it could be applied to any list already created

```
numbers = [4, -67, 12, 32, 14, -2, 7]
cubes = [x ** 3 for x in numbers]
print(cubes)
```

```
[64, -300763, 1728, 32768, 2744, -8, 343]
```

- We can also filter out some numbers
- For example, we might want to filter out negative values

```
numbers = [4, -67, 12, 32, 14, -2, 7]
cubes = [x ** 3 for x in numbers if x >= 0]
print(cubes)
```

```
[64, 1728, 32768, 2744, 343]
```

Example: increase all scores by 5%

```
print(scores)
scores_plus5 = [x + 5 for x in scores]
print(scores_plus5)
```

```
[90, 52, 69, 100, 89, 78, 95, 75]
[95, 57, 74, 105, 94, 83, 100, 80]
```

How do we avoid getting scores above 100?

- We could filter the list, but we need to be careful to not loose scores

```
print(scores_plus5)
scores_plus5_max100 = [x for x in scores_plus5 if x <= 100]
print(scores_plus5_max100)
```

```
[95, 57, 74, 105, 94, 83, 100, 80]
[95, 57, 74, 94, 83, 100, 80]
```

- We cannot put an `else` at the end of the list comprehension
- We have to move the `if ... else ...` at the beginning

```
scores_plus5_max100 = [x if x <= 100 else 100 for x in scores_plus5]
print(scores_plus5_max100)
```

```
[95, 57, 74, 100, 94, 83, 100, 80]
```

- The problem is that, in this way, we need 2 list comprehensions
- Better to use a function in a single list comprehension to get the correct score immediately

```
print(scores)
scores_plus5 = [min(x + 5, 100) for x in scores]
print(scores_plus5)
```

```
[90, 52, 69, 100, 89, 78, 95, 75]
[95, 57, 74, 100, 94, 83, 100, 80]
```

Dictionary Comprehensions

- Same principle as list comprehensions, but build a dictionary instead

Example: build a dictionary of cube values

```
numbers = [4, -67, 12, 32, 14, -2, 7]
cubes = {x: x ** 3 for x in numbers}
print(cubes)
```

```
{4: 64, -67: -300763, 12: 1728, 32: 32768, 14: 2744, -2: -8, 7: 343}
```

```
for k in sorted(cubes):
    # note: a better way to produce formatted output will be
    # covered later on
    print(k, "**3 =", cubes[k])
```

```
-67 **3 = -300763
-2 **3 = -8
4 **3 = 64
7 **3 = 343
12 **3 = 1728
14 **3 = 2744
32 **3 = 32768
```

- In this particular case, it might be better to create a list of points
- In Python, the points created with parentheses `()` are called *tuples*
- Tuples are exactly like lists, except that they are immutable (they cannot change)

```
numbers = [4, -67, 12, 32, 14, -2, 7]
cubes = [(x, x ** 3) for x in sorted(numbers)]
print(cubes)
```

```
[(-67, -300763), (-2, -8), (4, 64), (7, 343), (12, 1728), (14, 2744), (32, 32768)
```

Files

- Opening files for reading or writing is easy in Python
- But you still need to handle exceptions to make sure your program will not crash
- Use the `open` function to open a file
  - Give the file name (including path) as an argument
  - By default, it will open a file for reading
  - It will return a file object, to read from (or eventually write to)
- Better to close the file explicitely by calling the `close` function, even though the Python interpreter will eventually close it

```python
f = open("files/data.txt")
f.close()
```

Example: reading from a file, putting the lines into a list

```python
# version 1
lines = []
f = open("files/data.txt")
for line in f:
    lines.append(line)
f.close()
print(lines)
```

```
['12\n', '4\n', '5\n', '13\n', '-5\n', '3\n', '0\n', '4\n']
```

```python
# version 2
# call strip to remove whitespaces from beginning and end of lines
lines = []
f = open("files/data.txt")
for line in f:
    lines.append(line.strip())
f.close()
print(lines)
```

```
['12', '4', '5', '13', '-5', '3', '0', '4']
```

```python
# version 3
# in general, strip might remove too much, depending on the data
# only remove the \n instead
lines = []
f = open("files/data.txt")
for line in f:
    lines.append(line[:-1])
f.close()
print(lines)
```

```
['12', '4', '5', '13', '-5', '3', '0', '4']
```

```python
# version 4
# we can do much faster
with open("files/data.txt") as f:
    lines = f.readlines()
# with will automatically close the file
print(lines)
# how do we get rid of new lines?
```

```
['12\n', '4\n', '5\n', '13\n', '-5\n', '3\n', '0\n', '4\n']
```

```python
# version 5
# removing \n from lines
with open("files/data.txt") as f:
    lines = [line[:-1] for line in f.readlines()]
print(lines)
```

```
['12', '4', '5', '13', '-5', '3', '0', '4']
```

```python
# version 6
# convert each line to an int
with open("files/data.txt") as f:
    # the \n will not create problems here
    lines = [int(line) for line in f.readlines()]
print(lines)
```

```
[12, 4, 5, 13, -5, 3, 0, 4]
```

Example: writing the elements of a list to a file

```python
# version 1
# start with the data read in the previous example
with open("files/data2.txt", "w") as f:
    for line in lines:
        # must convert to string before writing
        f.write(str(line))
        # write will not automatically add \n after writing
        f.write("\n")
```

```python
# version 2
# use string formatting instead
with open("files/data2.txt", "a") as f:
    for line in lines:
        f.write("{}\n".format(line))
```

Exceptions

- Sometimes, unexpected things can happen in a program
- Many programming languages, such as Python, have the concept of *exceptions*
- When something goes wrong, an exception is thrown
- If not handled, the exception will end up crashing the program
- Use the `try ... except ...` construct to handle (or catch) exceptions

```python
numbers = [2, 5]
index = int(input("Index: "))
try:
    print(numbers[index])
except IndexError as err:
    print(err)
```

```
list index out of range
```

- In this particular case, using a `try ... except ...` is not the best strategy
- Exceptions should be for exceptional things: a user inputting an index out of range is quite common
- Use if statements instead

Exercise

- Rewrite the previous program without using exceptions handling
- Use some if statement instead

```python
numbers = [2, 5]
index = int(input("Index: "))
if 0 <= index < len(numbers):
    print(numbers[index])
else:
    print("index error")
```

```
index error
```

Example: exception handling when reading files

```
file_name = "files/data.txt"
try:
    with open(file_name) as f:
        lines = [int(line) for line in f.readlines()]
    print(lines)
except IOError:
    print("Problem opening file: " + file_name)
```

```
[12, 4, 5, 13, -5, 3, 0, 4]
```

Functions

- Defining functions in Python is easy: use the `def` keyword
- No types to declare
- Function parameters can have *default* values, therefore some parameters can be omitted when invoking a function
- Functions are *first-class objects* in Python

Example: function computing the absolute value of its argument

```
# define the abs function
def absolute(x):
    if x < 0:
        x = -x
    return x
```

- `x` is the only parameter to the function
- `x` has no default value, so an argument has to be provided to it when it is called

```
print(absolute(-5))
print(absolute(5))
```

```
5
5
```

Example: function with default value

```
# version 1
def increment(x, step=1):
    return x + step


counter = 1
print(counter)
print(increment(counter))
print(increment(counter, 3))
```

```
1
2
4
```

```
# version 2
def increment(x, step=1):
    x = x + step


counter = 1
print(counter)
increment(counter)
print(counter)
increment(counter, 3)
print(counter)
```

```
1
1
1
```

Questions

- Why do we get *1, 2, 4* as output in version 1?
- Why do we get *1, 1, 1* as output in version 2?

Passing Arguments to Functions

https://www.python-course.eu/python3_passing_arguments.php

- Correctly speaking, Python uses a mechanism, which is known as *Call-by-Object*, sometimes also called *Call by Object Reference* or *Call by Sharing*.
- If you pass **immutable arguments** like integers, strings or tuples to a function, the passing *acts like call-by-value*.
    - The object reference is passed to the function parameters.
    - They can't be changed within the function, because they can't be changed at all, i.e. they are immutable.
    - It's *different*, if we pass **mutable arguments**.
    - They are also passed by object reference, but they can be changed in place in the function.

`increment` function

- There is no need for an increment function
- Anyway, we can only change the value of parameter `x` locally
- Use the `+=` operator instead
- *Note*: there is no `++` operator in Python (use `+= 1` instead)

```
counter = 1
print(counter)
counter += 3
print(counter)
```

```
1
4
```

Python Functions are First-Class Objects

- Basically, this means that we pass around function references the same way we can pass object references
- We can, for example, maintain a list of function references, and call them on some arguments when looping through the list

```
# reuse the absolute function defined earlier,
# and the hex function used earlier
# put them in a list
# note that there are no () after the function names
# the () are used to call, or invoke, the functions
func_list = [absolute, hex]
```

```python
for func in func_list:
    print(func(-5312321))
```

```
5312321
-0x510f41
```

PyCharm Demo

- New project (pure Python, existing interpreter), Other project types later on.
- Project tab
- Python console
- Create new Python file
- Run a program
- Run configurations

Readings and Resources

1. The Python Tutorial
    - Sections 1 through 9
    - Sections 6 *Modules* and 9 *Classes* are useful but can be skipped for now
2. W3Schools Python Tutorial
    - *Python Tutorial*, all sections up to *Python Functions*
    - *Python Arrays* (not really arrays, but *lists*)
    - *File Handling*
    - *Python Classes* and *Python Inheritance* are useful but can be skipped for now