



**University of
Zurich** UZH

Foundations of Programming in Python

Philipp Gloor¹

¹University of Zurich

About me

Education

- 2012 – Bachelor of Science UZH in Physics
- 2016 – Master of Science UZH in Computational Science

Work

- 2014 – 2016 Software engineer CERN (remote)
- 2016 – now PDF Tools AG

Programming experience

C++, C#, Java, JavaScript, Python

Email

philipp.gloor@gmail.com

Round of introduction

- Name
- Occupation
- Programming experience? What language?
- Expectations

Learning targets

After this course...

- ... you will have an idea what programming is
- ... you will know how to write a basic computer program
- ... you will know the fundamental components of programming
- ... you are able to run Python code
- ... you are able to write a Python program based on a written out problem statement
- ... you know where you can find more information to improve your programming skills

Introduction to Programming

What is a Computer Program

Modular System

- **Input:** Data input from keyboard, files, internet, etc...
- **Output:** Processed data is displayed or saved to a file
- **Algorithms:**
 - **Assignment:** Values are assigned to variables
 - **Conditional execution:** Statements are executed only if certain conditions are fulfilled
 - **Loops:** Repeating statement or group of statements
- **Libraries:** Using existing implementations (can do anything of the above)

Examples: Hello World I

Java

```
public class HelloWorld {  
    public static void main(String args[]) {  
        System.out.println("Hello World");  
    }  
}
```

C++

```
#include <iostream>  
int main() {  
    std::cout << "Hello World\n";  
    return 0;  
}
```

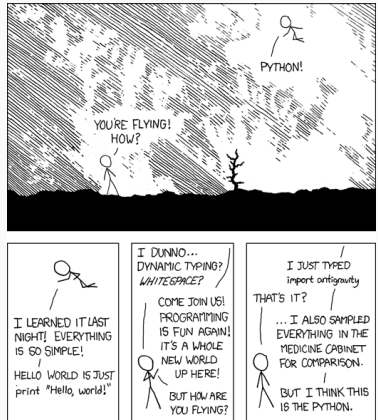
Examples: Hello World II

Python

```
print("Hello World")
```


Why Python?

- High-level programming language
- "Simple" syntax
- Cross-platform
- Interpreted (no compilation or linking needed)
- Object-oriented
- Many libraries available



Source: <https://xkcd.com/353/>

Assembly

Example of a low level language

```
.LC0:  
    .string "Hello world!"  
main:  
    push rbp  
    mov rbp, rsp  
    mov edi, OFFSET FLAT:.LC0  
    mov eax, 0  
    call printf  
    mov eax, 0  
    pop rbp  
    ret
```

How to Run Python Code

Options to run Python code:

- Directly in the Python prompt (REPL - Read, Eval, Print, Loop)
- Write the code into a file and run python with the file
- Use IDE to run Python code

Development Environment

- Integrated Development Environment (IDE)
- Collection of tools that are commonly used for software development (they make our life easier!)
- Popular IDEs
 - Visual Studio Code - <https://code.visualstudio.com>
 - JetBrains PyCharm - Community Edition available for free
<http://jetbrains.com/pycharm/download>
 - Eclipse with pydev - <http://pydev.org>

Fundamental Concepts

Types, Variables, Expressions, Operators, Comments

Types

- Numbers
 - 2
 - 1000000
 - -2
 - 3.2
 - 1.3333333
- Strings (Text)
 - 'Hello World'
 - "Hello World"

Data Types

Strings

- 'Single quotes' or "double quotes" can be used to declare them
 - 'Hello World'
 - "Hello World"
 - "5"

Boolean

Binary data type

- True
- False

Variables I

- Variables hold values
- Similar to mathematics
 - $x = 2$
 - $y = x + 2$
- Values assigned using the `=` operator

Variables II

Examples

Use meaningful names

- Declaration

```
salutation = "Hello"  
name = "Dennis Reynolds"  
pi = 3.14159
```

- Usage

```
print(name)
```

Variables III

Keywords - reserved words

```
and, assert, break, class, continue, def, del, elif, else, except, exec,  
finally, for, from, global, if, import, in, is, lambda, not, or, pass,  
print, raise, return, try, while, yield
```

Variables IV

Variables and values can be combined

```
print(2+2)
```

```
a = 2
```

```
print(a+2)
```

```
salutation = "Hello"
```

```
name = "Dennis Reynolds"
```

```
print(salutation + " " + name)
```

Operators

Order of precedence (kind of like PEMDAS)

- `()`
- `**`
- unary `+` `-`
- `*` `/` `%`
- binary `+` `-`
- `<`, `>`, `<=`, `>=`, `!=`, `==`
- **`not`**
- **`and`**
- **`or`**

Comments

- Comments have no impact on the program
- Should explain the code
- A comment starts with a # character

Examples

```
# Declaring the name  
name = "Philipp"  
print(name) # Prints Philipp
```

Functions I

- **print()** is a function that you have already used
- A function can take arguments which can be used inside the function

```
name = "Some name"  
print(name) # Some name is used inside the print function
```

- Functions can (and often do) also return a result
 - **return** statement

Examples

```
text = "Python programming language"  
print(text) # Prints: Python programming language  
text_length = len(text) # This function returns something  
print(text_length) # Prints length of the string
```

Functions II

Type conversions

- `int('32')`: Converts a string that holds a number to an integer
- `int('Hello')`: This doesn't work and it will throw a `ValueError` exception
- `float('313.333')`: Converts a string that hold a number to a float
- `str(32)`: Converts a number to a string

Examples

```
a = 20
b = 10
res = a + b
print("The sum of " + str(a) + " and " + str(b) + " is " + str(res))
```

Functions III

Rounding

```
a = 1.888
int(a) # = 1
int(round(a)) # = 2
int(a+5) # = 6
```

Math functions & First glance at a library

```
import math
log_res = math.log(17.0)
sin_res = math.sin(45)
angle = 20
x = math.cos(20*math.pi/180)) # cos/sin etc take radians as arguments →
    conversion from degree to radians necessary
```

- <http://docs.python.org/library/math.html>

Functions IV

User-defined functions

- A function encapsulates some functionality
- Reduces complexity

```
def print_two_values(param1, param2):  
    print(param1)  
    print(param2)
```

- Syntax is important
 - Indentation
 - The colon

Functions V

Examples

```
def line_separator():  
    print('')  
  
print("First Line")  
line_separator()  
print("Second Line")  
line_separator()  
print("Third Line")  
line_separator()  
print("Fourth Line")
```

- If we want to change the line separator to a dashed line we only need to change a single line of code

```
def line_separator():  
    print('_____')
```

Functions VI

Examples

- If the line separator should output two lines we can define a new function that calls the `line_separator()` function twice

```
def two_lines():  
    line_separator()  
    line_separator()  
  
print ("First Line")  
two_lines()  
print("Second Line")
```

Functions VII

Parameters and arguments

- Arguments are passed when calling a function
- Value of arguments is assigned to parameters

```
def print_sum(number_1, number_2):  
    result = number_1 + number_2  
    print(result)
```

```
print_sum(1,3)  
print_sum(10,5)
```

Functions VIII

Parameters and arguments

- Variables are valid within a scope
- Variables that are defined in a function can only be seen inside that function
- Scope can be identified by indentation

```
def concatenation(param1, param2):  
    concat = part1 + part2  
    print(concat)  
  
concatenation("Hello", "World")  
print(concat) # NameError: name 'concat' is not defined
```

Conclusion

- A function can be called multiple times
- If some code can be reused, put it in a function so you need to write less code
 - Higher factorization
 - Less redundancy
 - Better maintenance
- Functions can also call other functions

Naming Conventions I

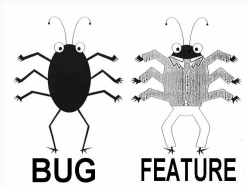
How to name your functions and variables (PEP8)

- Naming convention is a set of rules for choosing names of functions and variables
- Every programming language has different naming conventions
- Python
 - No spaces in variable and function names
 - Variable and function names are in lowercase and _ is used to separate words

```
length_in_cm = 15
```

```
def say_hello():  
    print("Hello")
```

Finding and resolving "bugs"



- Programming is a complex activity
- Mistakes happen all the time
- A mistake made in programming is called a bug
- The process of finding and resolving bugs is called debugging

Errors

- Syntax error
 - Incorrect syntax of a statement: `print(Hello World)` instead of `print("Hello World")`
- Runtime error
 - Error that occurs during the execution of a program
 - e.g. division by 0
- Semantic errors
 - Program does not deliver correct results
 - No error messages (code is syntactically correct)
 - Fixing semantic errors can be extremely complicated (good software design is important)

Techniques

- Reading code
- Print variables with `print()` to examine values (a poor man's debugger)
- Go through the program step by step -> **Debugger!**

Conditionals I

- Boolean algebra is a part of mathematics
- Often used in programming
- A boolean expression is either true or false

```
5 == 5 #  $\longrightarrow$  True
```

```
5 == 6 #  $\longrightarrow$  False
```

```
6 > 4 #  $\longrightarrow$  True
```

```
5 >= 8 #  $\longrightarrow$  False
```

Conditionals II

Examples

if

- The expression if defines a condition
- If the condition is true, subsequent statements will be executed
- If the condition is false, subsequent statements will not be executed
- There has to be at least one statement after the condition

```
x = 10
if x > 0:
    print(str(x) + ' is positive')
if True:
    # This statement will always be executed
    print('Yes')
if False:
    # This statement will never be executed
    print('No')
```

Conditionals III

else

- Expression else is executed if the if condition is false
- Can only be used in combination with an if expression

```
if x == 0:  
    print(str(x) + ' is zero')  
else:  
    print(str(x) + ' is not zero')
```

Conditionals IV

Examples

%-operator (remainder after division)

```
def print_parity(x):  
    if x % 2 == 0:  
        print(str(x) + ' is even')  
    else:  
        print(str(x) + ' is odd')
```

```
print_parity(2)
```

```
print_parity(3)
```

Chained conditionals

- `elif` is used to combine multiple conditions
- The `else` expression is executed when neither `if` nor any of the `elif`s is true.
- Any number of `elif` expressions can be used but only one `if` and one `else`

Conditionals VI

Examples

```
if x < y:
    print(str(x) + ' is less than ' + str(y))
elif x > y:
    print(str(x) + ' is greater than ' + str(y))
else:
    print(str(x) + ' and ' + str(y) + ' are equal')
```

```
# Python 3
answer = input('Do you like Python?')
# Python 2.7
# answer = raw_input('Do you like Python?')
if answer == 'yes':
    print('That is great!')
else:
    print('That is disappointing!')
```


Exercise 1

Solve exercise 1

Conditionals I

Nested conditionals

- Conditionals can be nested

```
if x > 0:  
    if x < 10:  
        print('x is a positive single digit')
```

and

- Deep nesting can be difficult to read
- Use **and** to combine conditionals

Conditionals II

```
if x > 0:
    if x < 10:
        print('x is a positive single digit')
# is the same as
if x > 0 and x < 10:
    print('x is a positive single digit')
```

or

- At least one statement must be true for the condition to be true

```
if x > 0 or x < 0:
    print("x is not zero")
```

Conditionals III

not

- Negation, inverts the boolean.
- **not** True -> becomes False
- **not** False -> becomes True

```
if not (y == 0):  
    print(x/y)  
else:  
    print("Cannot divide by zero")
```

X	Y	X and Y	X or Y
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Functions with return value I

- Some functions will return a value

```
# Python 3
```

```
answer = input('Do you like Python?')
```

```
# Python 2.7
```

```
# answer = raw_input('Do you like Python?')
```

- Our previously defined functions have never returned anything, but only printed something out

Functions with return value II

return

- Functions that return a value use the **return** keyword

```
import math
def area(radius):
    result = math.pi * radius ** 2
    return result

print(area(10))
my_circle_area = area(8)
```

- Functions can return any valid data type

Functions with return value III

Boolean return values

- The functions can return a boolean value (True, False)
- The function name should be formulated as a yes/no question

```
def is_divisible(x, y):  
    if x % y == 0:  
        return True  
    else:  
        return False
```

Functions with return value IV

Boolean return values

- The return value can be used in a condition

```
if is_divisible(x, y):  
    print(str(x) + ' is divisible by ' + str(y))  
else:  
    print(str(x) + ' is not divisible by ' + str(y))
```


Exercise 2

Solve exercise 2

Exercise 3

Solve exercise 3

Exercise 4

Solve exercise 4

Exercise 5

Solve exercise 5

Lists I

- Lists are a data type
- Lists are used in most programming languages (arrays)
- Lists are a set of values

```
list_a = [1, 2, 4]  
list_b = ['Monty', 'Python']
```

Creating lists

- The easiest way to create a list is using []

```
numbers = [10, 12, 14, 19]
words = ['spam', 'bungee', 'swallow']
```

- Data types can be mixed

```
my_list = ['music', 2000, 3.5, True]
```

Lists III

Creating lists

- Since numbers are often stored in a list, there is a special method for doing so
- With only one argument, range returns a number series starting at 0

```
list(range(4))  
# returns [0, 1, 2, 3]
```

- When using two arguments it is possible to define the start and end of the range [start, end) (end is not included in the list)

```
list(range(1,5))  
# returns [1, 2, 3, 4]
```

Creating lists

- The step size can be defined with a third argument

```
list(range(1, 10, 2))  
# return [1, 3, 5, 7, 9]
```

- An empty list can also be created

```
empty_list = []
```

- This is often done when the values to be inserted in the list are not yet known.

Creating lists

- Accessing elements can be done with the `[]` operator

```
names = ['Anna', 'Tom', 'Ralph', 'Peter']  
print(names[1])  
# prints Tom
```

Important

Array indices start at 0!

0		1		2		3
<hr/>						
Anna		Tom		Ralph		Peter

Accessing lists

- A negative index is used to access the list from the end

```
names = ['Anna', 'Tom', 'Ralph', 'Peter']  
print(names[-1])  
# prints Peter
```

Lists VII

Length

- The number of elements in a list can be obtained using the `len()` function

```
names = ['Anna', 'Tom', 'Ralph', 'Peter']  
print(len(names))  
# prints 4
```

Out of range

- If there is no item in the list at the desired index, Python will print an error message

```
names = ['Anna', 'Tom', 'Ralph', 'Peter']  
nNames = len(names)  
print(names[nNames])  
# IndexError: list index out of range
```

Changing elements in a list

- An element can be changed using [INDEX]

```
names = ['Anna', 'Tom', 'Ralph', 'Peter']  
names[0] = 'Alice'  
# ['Alice', 'Tom', 'Ralph', 'Peter']
```

Adding elements

- The `append()` method can be used to add an element at the end of the list

```
numbers = list(range(5))  
# [0, 1, 2, 3, 4]  
numbers.append(5)  
# [0, 1, 2, 3, 4, 5]
```

Concatenate lists

- The + operator can be used to join lists

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
# [1, 2, 3, 4, 5, 6]
```

Slices

- Lists can be cut into slices
- The operator `[n:m]` returns a list of the elements that start at index `n` and stop before `m`

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']  
my_list[1:3]  
# ['b', 'c']
```

Slices

- If the first index is empty, the slice starts at the beginning

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']  
my_list[:4]  
# ['a', 'b', 'c', 'd']
```

- If the second index is empty, the slice will include elements until the end of the list

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']  
my_list[3:]  
# ['d', 'e', 'f']
```


Deleting elements

- The `del()` method deletes items from the list

```
list_a = ['one', 'two', 'three']  
del(list_a[1])  
# ['one', 'three']  
list_b = ['a', 'b', 'c', 'd', 'e', 'f']  
del(list_b[1:5])  
# ['a', 'f']
```

Tuples I

Tuples is an immutable sequence data type

- It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.
- Tuples are declared using () instead of []

```
tuple = ('a', 'b', 'c', 'd', 'e')
```

- Tuples containing only one element must have a comma at the end of the definition

```
tuple = ('a', )
```

Strings I

Strings are immutable

- Unlike lists, strings cannot be changed
- Operations on strings always return a modified copy of the string
- The original string remains unchanged

```
greeting = 'Hello, world!'
greeting[0] = 'J'
# TypeError: 'str' object does not support item assignment
```

Iterations I

- Iterations are used to repeat statements
- There are two expressions for iterations
 - while
 - for

while

- As long as the condition of the while loop is True, the body of the loop gets executed

Iterations II

Example

```
def countdown(n):  
    while n > 0:  
        print(n)  
        n = n - 1  
    print('Lift off!')  
  
countdown(10)
```

Iterations III

while

- If the condition is False at the beginning, the body of the loop is never executed
- If the variable that is used to check the condition of the while loop does not change, the loop will never terminate -> infinite loop
- Whether a while loop terminates can be hard to determine

```
def sequence(n):  
    while n != 1:  
        print(n)  
        if n % 2 == 0:  
            n = n / 2  
        else:  
            n = n * 3 + 1
```

Iterations IV

while

- A **while** loop can be used to iterate through a list

```
names = ['Tom', 'Anna', 'Christopher']  
index = 0  
while index < len(names):  
    name = names[index]  
    print(name)  
    index = index + 1
```

Exercise 6

- Write a function `calc_sum(numbers)` which expects a list of numbers as input and returns their sum
- The method should be called as follows `calc_sum([4,6,7])`

Exercise 7

- Write a function `print_reverse(text)` which expects a string as an argument and prints every character of the string in reverse order
- Use a while loop to do this

Iterations

for

- Since it is often necessary to operate through lists and other data types, there is a special expression for this

```
for element in element_list:  
    print(element)
```

Exercise 8

Solve exercise 8

Key-Value pair

- Dictionaries are very similar to lists but have a key and value for each entry
- The entries of a dictionary are not sorted

Dictionaries II

Creating dictionaries

- Dictionaries are created using {}

```
eng2sp = {}  
eng2sp['one'] = 'uno'  
eng2sp['two'] = 'dos'
```

- Values can be added directly

```
inventory = {  
    'apples': 430,  
    'bananas': 312,  
}
```

Dictionaries III

Accessing entries

- Values can be accessed directly using `dictionary['key']`

```
inventory = {  
    'apples': 430,  
    'bananas': 312,  
}  
  
print(inventory['apples'])  
# 430
```

Dictionaries IV

Assigning and modifying values

- The key is assigned a value
- If the key already exists the existing value is overwritten

```
inventory = {  
    'apples': 430,  
    'bananas': 312,  
}  
inventory['oranges'] = 530  
inventory['bananas'] = 250  
print(inventory['bananas'])  
# 250
```

Deleting entries

- Key-Value pairs can be deleted using the `del()` function

```
inventory = {  
    'apples': 430,  
    'bananas': 312,  
}  
del(inventory['bananas'])
```


Number of entries

- The `len()` function returns the number of entries

```
inventory = {  
    'apples': 430,  
    'bananas': 312,  
}  
  
len(inventory)  
# 2
```

Dictionaries VII

Checking if an entry exists

- The `in` keyword can be used to check if a key exists in a dictionary

```
inventory = {  
    'apples': 430,  
    'bananas': 312,  
}  
  
if 'apples' in inventory:  
    inventory['apples'] += 100  
else:  
    inventory['apples'] = 100
```

Iterating over entries

- The `items()` function combined with the `for` statement can be used to iterate through every key-value pair

```
for (my_key, my_value) in my_dict.items():  
    print(my_key + ' : ' + my_value)
```

Exercise 9

Solve exercise 9

Exercise 10

Solve exercise 10

Exercise 11

Solve exercise 11

Persistence

Persistence

- So far no data has been saved in any of our examples
- All data was deleted from the memory as soon as our examples quit
- There are several ways to permanently store data on the hard disk
 - Database
 - Simple text files

Common procedure

- Open file
- Do something with the file
- Close the file

```
file = open('my_file.txt', 'mode')  
# do some stuff  
file.close()
```

Different modes

- The mode defines how the content of the file should be treated
- Modes
 - 'r': read only
 - 'w': write only
 - 'r+': read and write
 - 'a': append

```
# open a file in read/write mode  
file = open('my_file.txt', 'r')
```

Write

- The `write()` function is used to write something into a file
- `'\n'` is used to insert a line break

```
file = open('my_file.txt', 'a')
file.write('Das ist eine Linie\n')
file.write('Das ist eine neue Linie\n')
file.close()
```

Files IV

Read

- A `for` loop can be used to read a file line by line
- `line.strip()` removes the trailing `'\n'`

```
file = open('my_file.txt', 'r')
for line in file:
    line = line.strip()
    print line
file.close()
```

Dictionaries/list in JSON

- `file.write()` only accepts strings as arguments
- If complex structures such as dictionaries or lists should be stored in a file, it's necessary to convert these structures into strings first
- An example of a standard used for this purpose is JSON (Javascript Object Notation)

```
import json
my_dict = {'one': 'uno', 'two': 'dos'}
my_dict_as_string = json.dumps(my_dict)
print(my_dict_as_string)
```

Convert JSON to dictionaries/lists

- Example of a string in JSON that is converted into a dictionary

```
import json
my_dict_as_string = '{"two": "dos", "one": "uno"}'
my_dict = json.loads(my_dict_as_string)
print(my_dict)
```

Exercise 12 I

Solve exercise 12

Additional Resources

- How to Think Like a Computer Scientist from Allen Downey, Jeffrey Elkner, and Chris Meyers
- Learning with Python: Interactive Edition 2.0
 - <http://interactivepython.org/courselib/static/thinkcspy/index.html>
- Official Python Documentation
 - <http://www.python.org/doc/>
- Project Euler: Mathematical problems that can be solved programmatically
 - <http://projecteuler.net/>
- Platforms to prepare for coding interviews
 - <https://leetcode.com>
 - <https://www.interviewbit.com/>