University of Zurich^UZH

Zentrale Informatik

Foundations of Programming in Python

David Pinezich

About me

- Education
  - Informatiker Applikationsentwicklung EFZ (BMS / Passerelle)
  - Bachelor of Informatics at UZH
  - Currently – Master of Informatics at UZH

- Work Experience
  - Paul Scherrer Institut (PSI)
  - Architonic
  - ti&m
  - Helsana (Lead Webengineering)

- Programming Experience

- david.pinezich@gmail.com / david.pinezich@uzh.ch

What about you?

- Name
- Field of Study
- Do you have any programming experience?
  - Languages / Projects?
- Expectations for this course?
- Special requests?

Learning Targets

- After this course…
- ... you will know 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 some 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

- **And nevertheless you will confess your friends that programming is fun!**

Course Overview

- Introduction to Programming
- Fundamental Components
  - Values, Variables, Expressions, Operators, Comments
  - Functions
  - Conditionals
  - Functions with return values
  - Lists
  - Iterations
  - Dictionaries
- Persistence

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
- **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

Examples – Hello World

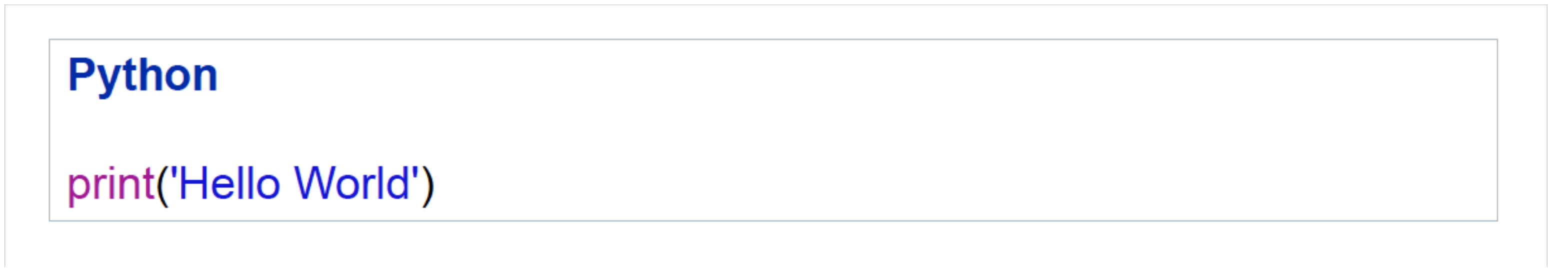
Java

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

C++

```
#include <iostream>
using namespace std;
int main() {
    cout << 'Hello World!' << endl;
    return 0;
}
```

Examples – Hello World



Python

```python
print('Hello World')
```

Why Python

- Simple syntax
- High-level programming language
- Cross-platform
- Interpreted
- Object-oriented
- Many libraries available

Development Environment

- Integrated Development Environment (IDE)
- Collection of tools that are commonly used for software development
- Popular IDEs
  - Eclipse with Python Plugin (PyDev): http://pydev.org/
  - JetBrains PyCharm: http://www.jetbrains.com/pycharm/
    - Community Edition is available for free: http://www.jetbrains.com/pycharm/download/

Demo time! Hello World

Options to run Python code

- Directly in the terminal
- Save Python-Code to a file and run it directly
- Use an IDE to run Python-Code
- Run it in Jupyter Notebook

```
 # -*- coding: utf-8 -*-
print('Hello World')
```

But WAIT! How? Let’s try together!

- With your terminal (Mac OS)
  - Magnifying glass → Type “Terminal” → Hit enter
  - Type python & hit enter again
  - Code ☺
- Run directly
  - Save hello.py on your desktop
  - Open the Terminal again
  - Run >> python /Users/***dave***/Desktop/hello.py
- With Pycharm (harder to understand, easier for later)
  - https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html.html
- Jupyter Notebook (web or local)
  - Put the code in and hit run

Python 3 or 2?

- Python 3 because it is state of the art
- References to Python 2 at some occasions
  - Because some tools are not yet migrated

A very good overview:
http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

Videos & Additional learning resources

- Lynda.com
  - https://www.lynda.com/Python-tutorials/Programming-Fundamentals-Real-World/418249-2.html?org=uzh.ch
  - https://www.lynda.com/Python-tutorials/Python-3-Essential-Training/62226-2.html?org=uzh.ch
  - Example:
    - https://www.lynda.com/Python-tutorials/Ifelse-chains/418249/459142-4.html?org=uzh.ch

Fundamental Components

Values, Variables, Expressions, Operators, Comments

Values

- Numbers
  - 2
  - 1000000
  - -2
  - 3.2
  - 4.23333333
- Strings (Text)
  - 'Hello World'
  - "Good morning"

Data Types

Values have data types

Numbers

- Integers: *No dot*
  - 3
  - -5
  - 10000000
- Floats: *Have a dot*
  - 23.222
  - 3.0
  - -2.2

Data Types

Strings

- Either '' or "" are used to declare them
    - 'Hello World'
    - "Hello World"
    - "5"

Boolean

- Binary expression
    - *True (1)*
    - False (0)

Variables

- Hold values
- Similar to mathematics
  - X = 2
  - Y = X + 2
- Values are assigned using the = operator

Variables

Examples

- Use meaningful names
- Declaration

```
salutation = 'Hello'
name = 'Monty Python'
pi = 3.14159
```

- Usage

```
print(name)
```

Variables

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

Variables and values can be combined

- print(2 +2)
- a = 2
- print(a + 2)

- salutation = 'Hello'
- name = 'Monty Python'
- print(salutation + ' ' + name)

Operators

Order of precedence

- ()
- ** (exponential - 2**4 = 16)
- *, /, %, //
- +, -
- <<, >>, &, ^, | (bitwise operations)
- <, > <=, >= (comparison)
- < >, !=, == (equality)
- =, %=, /=, //=, -=, +=, *=, **= (assignment)
- is, is not (identity)
- in, not in (membership)
- not, or, and (logical)

Comments

- Comments have **no** impact on the program
- Should explain the code (if necessary)
- A comment starts with the # sign

```python
# The following line declares the name
name = 'David'
print(name) # David is printed out
```

Fundamental Components

Functions

Functions

- print() is a function that you have already used already
- A function can take arguments, which are values that are bound to variables **inside the function**

```
name = 'Monty Python'
print(name) # Monty Python is printed out
```

- Functions can also return a result
  - return statement

Functions

Example

```
text = 'Python programming language'
print(text) # Python programming language is printed out
text_length = len(text)
print(text_length) # 27 is printed out
```

Functions

Type conversions

- int('32')
  - Converts a string that holds a number to an integer
- int('Hello')
  - Only a string that hold an integer can be converted to an integer
    - ValueError: invalid literal for int() with base 10: 'Hello‘
- float('313.333')
  - Converts a string that holds a decimal number to a float

Functions

Type conversions

- str(32)
- Converts a number to a string

Example

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

Functions

Rounding

Example

```
a = 1.888
int(a) # Equals 1
int(round(a)) # Equals 2
int(a + 0.5) # Equals 2
```

Functions

Math Functions

Example

```
import math

log = math.log(17.0)
sinus = math.sin(45)
angle = 20
x = math.cos(angle + math.pi/2)
```

http://docs.python.org/library/math.html

Functions

User-Defined Functions

- A function encapsulates some functionality
- Reduction of complexity and redundancy

Example

```
def my_function(parameter1, parameter2):
    print(parameter1)
    print(parameter2)
```

- Syntax is **very** important!
    - Indentation is necessary (4 spaces or 1 tab)
    - Don’t forget the colon (not the semi-colon)

Functions

User-Defined Functions – Example

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

print('First Line')
line_separator()
print('Second Line')
line_separator()
print('Third Line')
line_separator()
print('Fourth Line')
```

Functions

User-Defined Functions

- 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('------------------------------')
```

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

Functions

User-Defined Functions – Example

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

```
def two_lines():
    line_separator()
    line_separator()

print ('First Line')
two_lines()
print('Second Line')
```

Functions

User-Defined Functions – Conclusion

- A function can be called multiple times
- Less code needs to be written if functions are used multiple times
  - Higher factorization
  - Less redundancy
  - Better maintenance
- Functions can also call other functions

Functions

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

Parameters and Arguments

- Variables are valid within a scope
- Variables that are defined in a function are only valid in the function’s scope
- Indentation helps to identify scope

```
def concat_twice(part1, part2):
    concat = part1 + part2
    print(concat)

concat_twice('Hello','World')
print(concat)
# NameError: name 'concat' is not defined
```

Naming Conventions & Debugging

Naming Conventions

How to name your functions and variables

- 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')
```

Debugging

Finding and resolving Defects

- 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

Debugging

Errors

- **Syntax Error**
  - Incorrect syntax of a statement
  - e.g. print(Hello World) instead print('Hello World')

- **Runtime Error**
  - Error that occurs during the execution of a program
  - e.g. division by 0

Debugging

Errors

- **Semantic Error**
  - Program does not deliver correct results
  - No error messages
  - Fixing semantic errors can be extremely complicated

Debugging

Techniques

- Reading code (multiple times)
- Print variables to examine the values
    - e.g. using print(variable)
- Go through program step by step
    - Debugger Tool
- Rubber Duck Debugging
    - https://en.wikipedia.org/wiki/Rubber_duck_debugging

Fundamental Components

Conditionals

Conditionals

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

```
5 == 5 # --> True
5 == 6 # --> False
6 > 4 # --> True
5 >= 8 # --> False
```

Conditionals

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

Conditionals

if

```
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

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

Modulo Operator – Example

```
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)
```

Conditionals

Chained conditionals

- Expression elif is used to combine multiple conditions
- The else expression is executed at the end if neither the if condition nor a single elif condition is true
- Any number of elif expressions can be used, but only one if and one else statement

Conditionals

Chained conditionals

```
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')
```

Conditionals

User input – Example

```
# 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

- Write a function compare(x,y) that
  - prints out 1 if x is greater than y
  - prints out 0 if x is equal to y
  - prints out -1 if x is less than y
- input() should be used to receive the numbers as user input
  - By using python 2.7 it is raw_input()

Exercise 1 – Additional info

- Attention: input() stores the input as a string (not as a number!)
- If the input should be a number (integer or float), it must be converted

```
first_number = input('Please enter a first number ')
first_number = int(first_number)
second_number = input('Please enter a first number ')
second_number = int(second_number)
result = first_number + second_number
print(str(result))
```

Conditionals

Nested conditionals

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

Conditionals

and

- Deep nesting often difficult to understand
- Can be combined with and statement

```
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')
```

Conditionals

or

- At least one statement must be true for the condition to be true
- If both statements are true, the condition is also satisfied

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

Conditionals

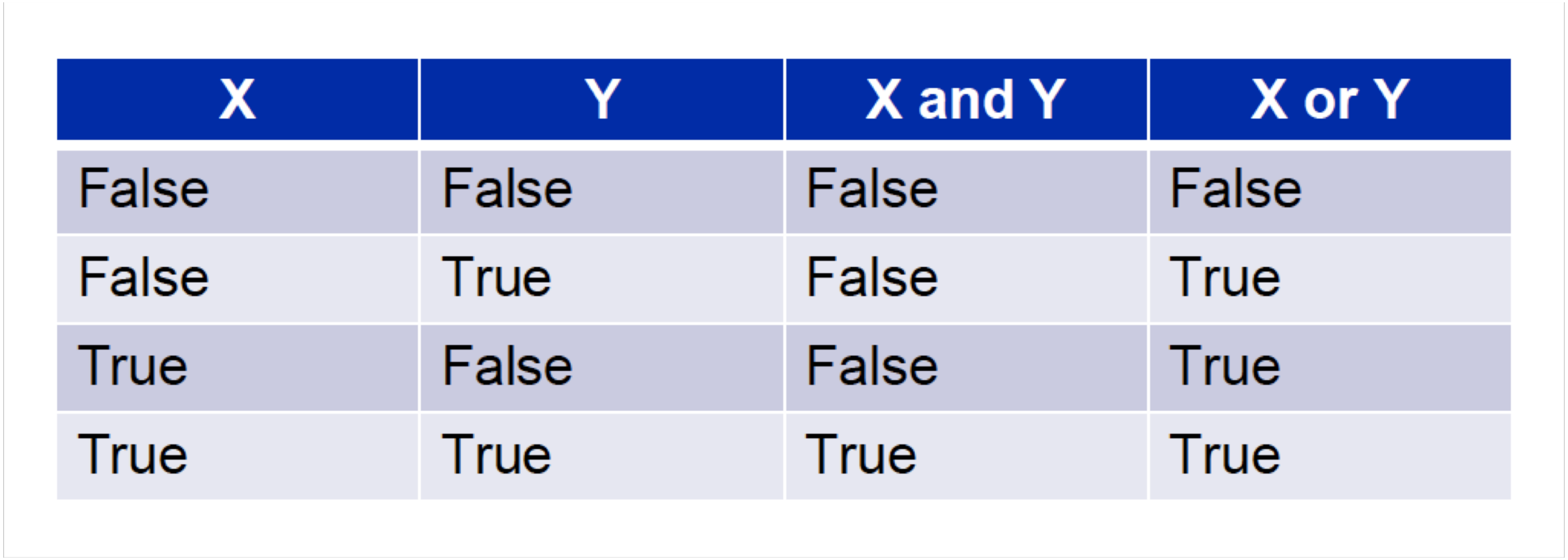
not

(not True) becomes False

(not False) becomes True

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

Conditionals



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

Fundamental Components

Functions with return values

Functions with return values

- 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 values

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 values

Boolean return values

- 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 values

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

- Write a function volume_from_radius(radius), which calculates the volume of a sphere
- Note
  - Pi is math.pi
  - Attention when dividing 4 / 3
  - 4 / 3 = 1
  - 4.0 / 3.0 = 1.33333333
  - Use import math statement at the beginning of the file
  - $$volume = \frac{4}{3}\pi * radius^3$$

Exercise 3

- Write a function called distance(x1, y1, x2, y2) which computes the distance between point1 (x1, y1) and point2 (x2, y2)
- Note
  - X^2 is represented by x**2 in Python
  - The root of x is computed with math.sqrt(x)
  - Use import math statement at the beginning of the file
  - $$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Exercise 4

- Write a function volume_from_points(x1, y1, x2, y2)
- This function calculates the volume of a sphere whose radius is the distance between the points (x1, y1) and (x2, y2)
- Tip: Use the implemented methods from the previous exercises

Exercise 5

- Write a function is_between(x, y, z) which returns true if x <= y <= z and False otherwise

Fundamental Components

Lists

Lists

- 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']
```

Lists

Creating lists

- The easiest way to create is using []

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

- Data types can be mixed

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

Lists

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]
```

- Using two arguments it’s possible to define start and end
- The second argument defines the last number that is not part of the list

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

Lists

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

```
empty_list = []
```

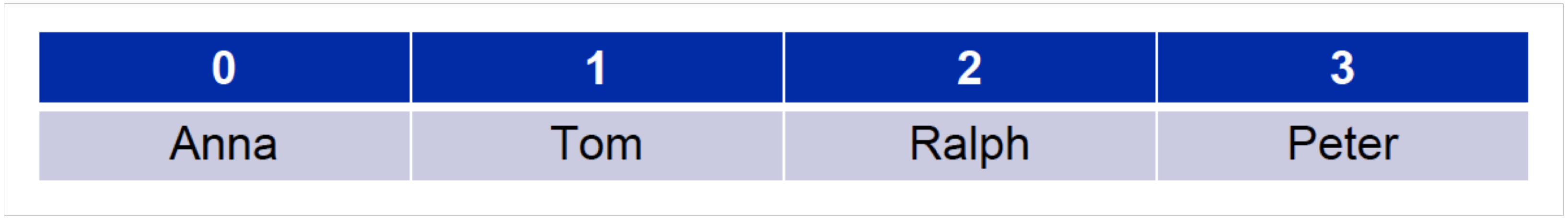
Lists

Creating lists

- [INDEX] is used to access an element in a list

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

- Important: Numbering of indices always starts at 0!



0	1	2	3
Anna	Tom	Ralph	Peter

Lists

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

Length

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

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

Lists

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']
list_length = len(names)
print(names[list_length])
# IndexError: list index out of range
```

Lists

Changing elements in a list

- An element can be changed using [INDEX]

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

Lists

Adding elements

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

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

Lists

Concatenate lists

- The + operator can be used to concatenate lists

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

Lists

Slices

- Lists can be cut into slices
- The operator [n:m] returns a list of the elements that starts at index n and stops before m
  - The element at index n is present in the list, the element at index m is not though

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

Lists

Slices

- If the first index is blank, 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 blank, the slice will include all subsequent elements including the last one

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

Lists

Deleting elements

- The del() method can be used to delete items from a 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

Tuples are immutable lists

- Tuples are the same as lists, except the fact that they can not be changed
- Tuples are declared using ( ) instead of [ ]
- Tuples must not contain exactly two elements

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

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

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

Strings

Strings are immutable

- Unlike lists, strings can not 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
```

Fundamental Components

Iterations

Iterations

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

Iterations

while

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

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

countdown(10)
```

Iterations

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**

Iterations

while

- Whether a while loop terminates or 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

while

- A while loop can be used to iterate through a list
- In this case, every element of the list is printed out

```
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 an argument and returns their sum.
- The method should be called as follows:
    - calc_sum([4,6,10])

Iterations

for

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

```
for my_element in my_list:
    print(my_element)
```

- The for loop iterates through every element in a list

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

Exercise 8

- Write a function count_words(words, min_word_length) that counts the number of words in a list that are at least as long as the specified word length
- Use a for loop to do this
- Example:

```
words = ['Emanuel', 'John', 'Ale']
count_words(words, 4)
# 2
```

Fundamental Components

Dictionaries

Dictionaries

Dictionaries are key-value pairs

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

Dictionaries

Creating dictionaries

- Dictionaries are created using {}

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

- Values can be added directly

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

Dictionaries

Accessing entries

- Values can be accessed using dictionary_name['key']

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

Dictionaries

Assigning and modifying values

- The key is assigned a value
- If the key already exists, the value of the corresponding key is modified

```
inventory = {
    'apples': 430,
    'bananas': 312,
}
inventory['oranges'] = 530

inventory['bananas'] = 250
print(inventory['bananas'])
# 250
```

Dictionaries

Deleting entries

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

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

Dictionaries

Number of entries

- The len() function returns the number of entries

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

Dictionaries

Checking if an entry exists

- The in keyword can be used to check if an entry exists

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

Dictionaries

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 – Part 1

- Write a function calculate_mark(points, max_points) which returns a grade in the Swiss grading scale

$$mark = \frac{points \times 5}{max\_points} + 1$$

- The function rounds the grade to the nearest 0.5
  - 5.6666 –> 5.5
  - 5.75 → 6
  - mark_rounded = round(mark * 2) * 0.5
- The function should accept strings as arguments
  - Arguments should therefore be converted to floats

Exercise 9 – Part 2

- Write a function that asks for points and max_points as long as the user does not enter “exit”
- The grade should be printed out after each run
- Idea:

```
while True:
    # input points (use input)
    if points == 'exit':
        break
    # input max_points
    # call calculate_mark function
    # print result
```

Exercise 9 – Part 3

- Change your code that it additionally asks for a name
- A dictionary should now store the grade of each name
    - The name is the key, the grade the value
- As soon as the user enters “exit”, the program should print out the grades of all names before it quits

Exercise 9 – Part 4

- Change your code in such a way that for each name it additionally outputs if the user has passed or failed
  - mark >= 4 → passed
  - mark < 4 → failed

Exercise 9 – Part 5

- Change your code in such a way that the application outputs the average grade before it stops

Exercise 10

- Write an application that generates a random number between 1 and 100
    - import random
    - random.randrange(min, max)
- The user makes a guess and enters a number. If the number is incorrect, the program outputs whether the entered number was too small or too large and allows the user to guess again.
- The application quits when the correct number is guessed
- The application should output how many user attempts have been made before it quits

Exercise 11

- Implement the opposite of Task 10 so that the user thinks of a number and makes the computer guess
- The user provides feedback on whether the number is too high, too small, or correct
  - < (too low)
  - > (too high)
  - = (correct)
- How many steps does the computer need?

Persistence

Persistence

Saving data

- 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

Persistence – Files

Common procedure

- 3 Steps:
  - Open file
  - Do something with the file
  - Close file

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

Persistence – Files

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

```
file = open('my_file.txt', 'mode')
```

Persistence – Files

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')
file.write('Das ist eine neue Linie')
file.close()
```

Persistence – Files

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()
```

Persistence – JSON

Dictionaries/lists 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 the convert these structures into strings first
- An example of a standard used for this purpose is JSON (Javascript Object Notation)

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

Persistence – JSON

Convert JSON back 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 – Part 1

- Write an application which repeatedly asks for a name and phone number until the user enters “exit”
- Each name/telephone number pair should be stored as an entry in a dictionary
  - The names are the keys of the dictionary
  - The telephone numbers are the values of the dictionary
- As soon as the user enters “exit”, create a JSON string of the dictionary using the json.dumps() function and store the string in a file called address_book.txt

Exercise 12 – Part 2

- Extend your application so that it reads the address_book.txt file when it starts
- Convert the JSON text into a dictionary again

```
import json
address_book_file = open('address_book.txt', 'r')
address_book_dict = json.load(address_book_file)
```

- Ask the user if he wants to add more names or not
- Let the user search for names in the dictionary and print out the according phone number

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/