

# Programming for beginners: Python

LET-CIWB269-IBC

## Lecture 1: Introduction

# Overview of today's introduction

- Introduction of Eric Sanders
- Attendees
- General information about the course
- Introduction of Cristian Tejedor Garcia
- Basic information about computers and programming

# Eric Sanders

MA Language and Speech Technology

(Nijmegen 1995)

Worked a lot on (speech databases for)  
development of ASR

Current project: predict election outcome  
based on Twitter

Started programming 35 years ago with BASIC  
Python main programming language for many years now

# Attendees



# General information about course

2 periods (6 Sep - 22 Oct , 8 Nov - 24 Dec)

Lecture on Monday 12:30-13:15 (Cristian)

Practical on Friday 8:30-10:15 (Eric)

Beginners course! We assume no prior programming knowledge

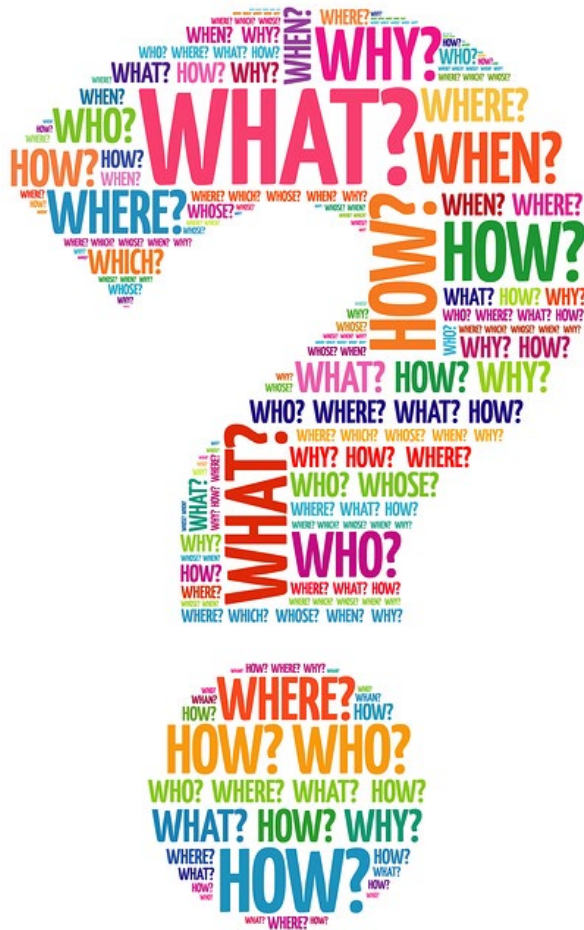
Goal: getting insight into when you can use automatisations for (natural language processing) experiments and make a start with python programming

Schedule: on Brightspace, subject to change

Book: Think Python, freely available online

# Assignments & Examination

Before each practical session assignments are on Brightspace  
Assignments will be made during practical sessions and at home  
Assignments have to be handed in in five days (Wednesday 9:00)  
Checked but no individual feedback (questions during practicals)  
After first period, bigger assignment will be handed out for which feedback will be given  
At the end, take home exam (questions + programming tasks)  
If weekly assignments are sufficient, take home exam decides grade.



# Cristian Tejedor García

<https://www.cristiantg.com>

PhD in Computer Science

(University of Valladolid, 2020)

Design and evaluation of mobile  
computer-assisted pronunciation training  
tools for second language learning

Current projects at CLST: ST.CART and HoMed:

Dutch ASR systems for children and elderly people

Started programming 11 years ago in Pascal & C  
and 10 years ago in Python



# What is a computer?

CPU (brain)

RAM (active memory)

Hard disk (long term memory)

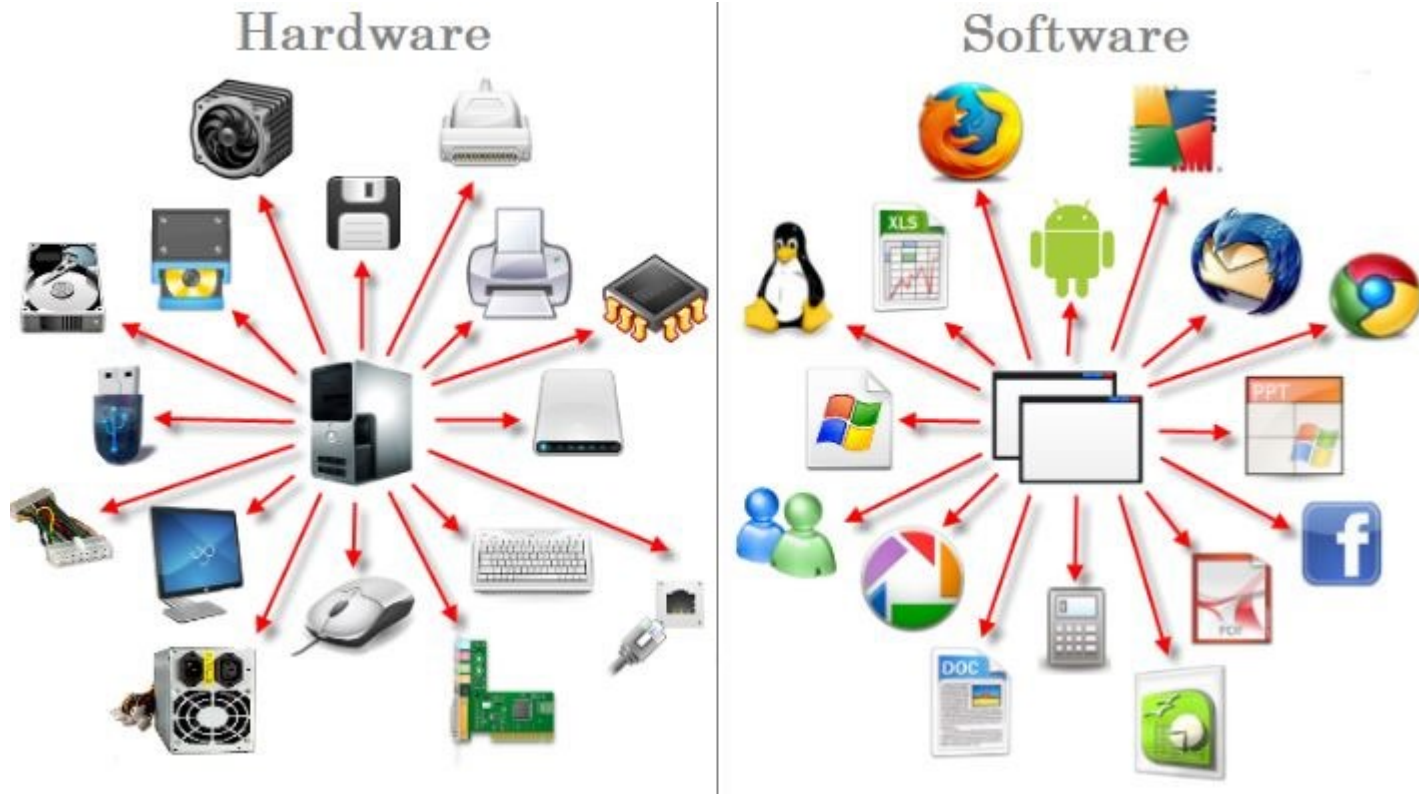
IO

Input (senses) like keyboard, mouse, microphone

Output (communication) like monitor, printer, speakers



# Hardware vs Software



# Operating system

Layer between **hardware**  
and **software**



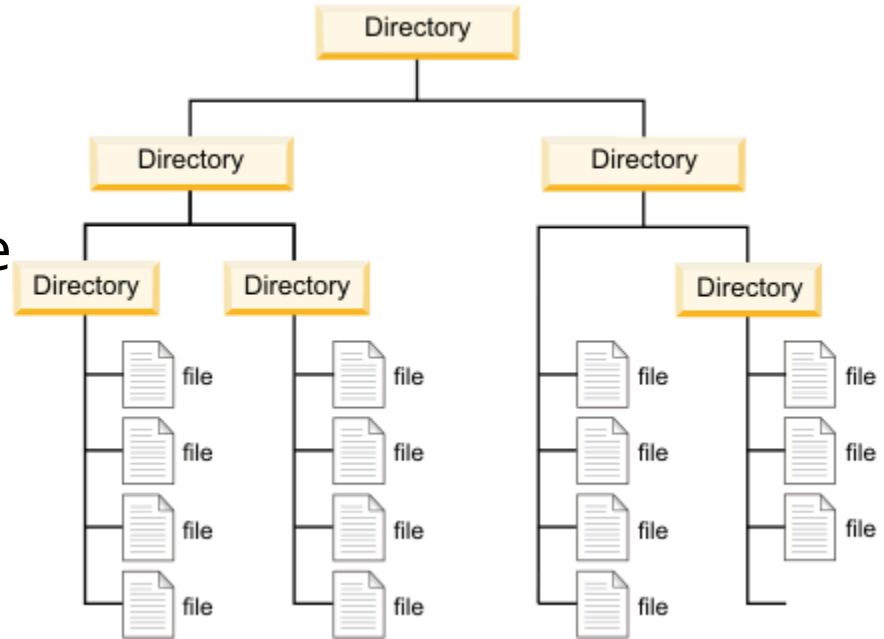
# File system

Directory (branch):

contains directories & file

File (leaf):

actual data or program  
either text or binary





# Computer program

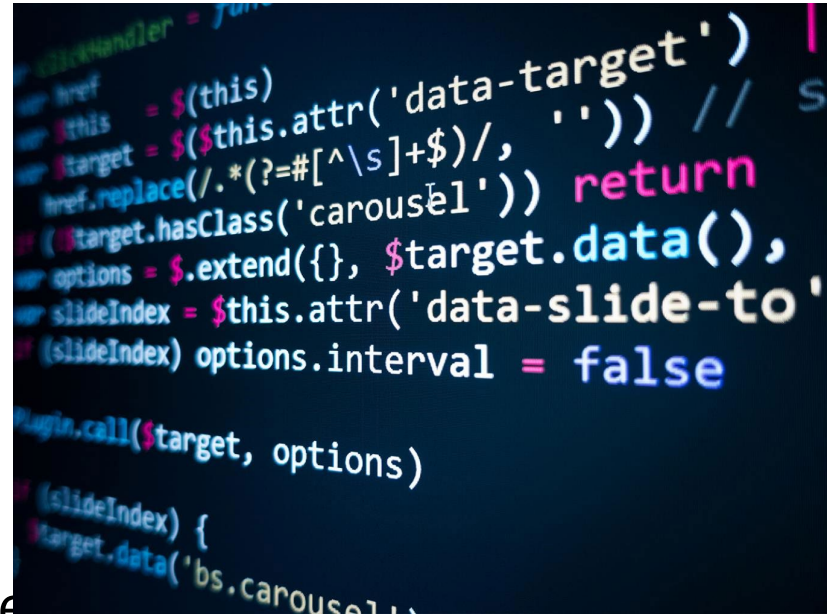
Set of instructions that tells a  
computer what to do

Compare to recipe, but with very  
strict syntax

Everything a computer does is  
programmed:

OS, webbrowser, music player,

email program, etc., etc. => many programmers needed



```
clickHandler = function() {  
    var href = $(this).attr('data-target')  
    var target = $(this.attr('data-target') // s  
    href.replace(/.*(?=#[^\s]+$)/, ''))  
    if (!target.hasClass('carousel')) return  
    var options = $.extend({}, target.data(),  
    { slideIndex: $(this.attr('data-slide-to'  
    (slideIndex) options.interval = false  
    plugin.call(target, options)  
    (slideIndex) {  
        target.data('bs.carousel',
```

# Programming languages

Many programming languages  
(google says 700)

Best for low level: C or C++

Best for web applications: Java

Best for back-end: php, javascript

Best for front-end: javascript

Best for statistical analysis: R

Best to start with...



# Python

Developed by Guido van Rossum (NL) in 1990

Named after Monty Python

Much used in academia (but also industry)

Very suitable for text/file processing

Much support (many modules)

Readable and maintainable

Easy to start with but still usable for professionals (Instagram and Youtube are programmed in Python)



# Lecture 2:

# Variables & Data types

Programming for beginners: Python

# What is a program?

Sequence of instructions that specifies how to perform a computation

# What is a program made of?

- **input:** Get data from the keyboard, a file, the network, or some other device
- **output:** Display data on the screen, save it in a file, send it over the network, etc.
- **math:** Perform basic mathematical operations like addition and multiplication
- **conditional execution:** Check for certain conditions and run the appropriate code
- **repetition:** Perform some action repeatedly, usually with some variation

# Example: Using python as a calculator

**Exercise 1.2.** *Start the Python interpreter and use it as a calculator.*

1. *How many seconds are there in 42 minutes 42 seconds?*
2. *How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.*
3. *If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?*

# Example: Using python as a calculator

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> (42*60)+42
```

```
2562
```

```
> 10/1.61
```

```
6.211180124223602
```

```
> 
```



# Example: Using python as a calculator

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> (42*60)+42
```

```
2562
```

```
> 10/1.61
```

```
6.211180124223602
```

```
> 2562/6.211180124223602
```

```
412.482
```

```
> 
```

# Example: Using python as a calculator

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

> (42\*60)+42

2562

> 10/1.61

6.211180124223602

> 2562/6.211180124223602

412.482

> 60\*60

3600

> 3600/412.482

8.727653570337614

>

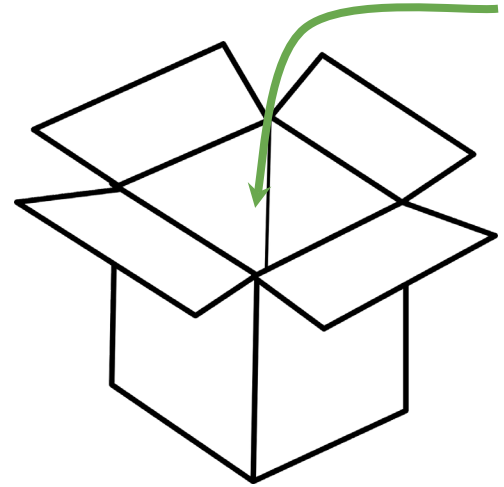
What could have made this  
process more efficient?

# Variables!

A variable is a name that refers to a value

# What are variables good for?

- Allow you to **reuse** things you have computed earlier
- Think of it as a box to **store** something in, so that you can reuse it later
  - **Note:** Value stored in the variable can be overwritten
- Can be used like the  $x$  in a math equation: a **placeholder**



## Example: Exercise 1.2 (part 3) with variables

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> time_in_sec = (42*60)+42
```

```
> print(time_in_sec)
```

```
2562
```

```
> miles_in_10_km = 10/1.61
```

```
> print(miles_in_10_km)
```

```
6.211180124223602
```

```
> 
```

## Example: Exercise 1.2 (part 3) with variables

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> time_in_sec = (42*60)+42
```

```
> print(time_in_sec)
```

2562

```
> miles_in_10_km = 10/1.61
```

```
> print(miles_in_10_km)
```

6.211180124223602

```
> avg_pace = time_in_sec/miles_in_10_km
```

```
> print(avg_pace)
```

412.482

```
> 
```

## Example: Exercise 1.2 (part 3) with variables

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> time_in_sec = (42*60)+42
```

```
> print(time_in_sec)
```

2562

```
> miles_in_10_km = 10/1.61
```

```
> print(miles_in_10_km)
```

6.211180124223602

```
> avg_pace = time_in_sec/miles_in_10_km
```

```
> print(avg_pace)
```

412.482

```
> hour_in_sec = 60*60
```

```
> print(hour_in_sec)
```

3600

```
> █
```



## Example: Exercise 1.2 (part 3) with variables

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> time_in_sec = (42*60)+42
```

```
> print(time_in_sec)
```

2562

```
> miles_in_10_km = 10/1.61
```

```
> print(miles_in_10_km)
```

6.211180124223602

```
> avg_pace = time_in_sec/miles_in_10_km
```

```
> print(avg_pace)
```

412.482

```
> hour_in_sec = 60*60
```

```
> print(hour_in_sec)
```

3600

```
> avg_speed = hour_in_sec/avg_pace
```

```
> avg_speed
```

8.727653570337614

```
> 
```

# Values and types

## Values we've seen so far:

- "Hello, World!" →
- 2562
- 3600
- 6.211180124223602
- 412.482
- 8.727653570337614

= string

= integer

= float (i.e. floating-point number)

# Values and types

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> type("Hello, World!")
```

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

```
> type(2562)
```

```
<class 'int'>
```

```
> type(412.482)
```

```
<class 'float'>
```

```
> █
```

# Values and types

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> type("2562")
```

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

```
> type('412.482')
```

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

```
> []
```

# Variables can contain any data type

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> planet_greeting = "Hello, World!"
```

```
> miles = 10
```

```
> km = miles*1.61
```

```
> type(planet_greeting)
```

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

```
> type(miles)
```

```
<class 'int'>
```

```
> type(km)
```

```
<class 'float'>
```

```
> 
```

# Typecasting

- Can be used to explicitly change type of given variable or value

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)  🔍 ✕
> miles = 10
> type(miles)
<class 'int'>
> miles_as_float = float(miles)
> print(miles_as_float)
10.0
> type(miles_as_float)
<class 'float'>
> km = miles*1.61
> print(km)
16.1
> type(km)
<class 'float'>
> km_as_int = int(km)
```

# Typecasting

```
➤ km = miles*1.61
➤ print(km)
16.1
➤ type(km)
<class 'float'>
➤ km_as_int = int(km)
➤ print(km_as_int)
16
➤ type(km_as_int)
<class 'int'>
➤
```

# Typecasting

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> miles = 10  
> km = miles*1.61  
> type(km)  
<class 'float'>  
> km_as_string = str(km)  
> print(km_as_string)  
16.1  
> type(km_as_string)  
<class 'str'>  
> km_as_string / 1.61
```



# Typecasting

```
> km_as_string = str(km)
```

```
> print(km_as_string)
```

```
16.1
```

```
> type(km_as_string)
```

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

```
> km_as_string / 1.61
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for /: 'str' and  
d 'float'
```

```
> █
```

# Variable assignment & naming

- **How to assign a variable:**

- km = 10
- first\_name = "Monty"

} variable name = value

- **Variable naming rules:**

- Variable names can contain both letters and numbers, but they can't begin with a number
- Uppercase letters allowed, but conventional to use only lower case
- Underscore is often used to separate words in multi-word variable names
- Cannot use Python's own keywords as variable names

# Python keywords (Python 3.8.5)

## 2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

# Python keywords (Python 3.8.5)

main.py

```
1 leave = "Goodbye!"
2 print(leave)
3
4 return = "Hello again!"
5 print(return)
```

<https://Hello-World.mariekewoe.repl.run>

File "main.py", line 4  
return = "Hello again!"

^  
SyntaxError: invalid syntax

# Overwriting contents of a variable

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> first_name = "Monty"
> last_name = "Python"
> full_name = first_name + last_name
> full_name
'MontyPython'
> first_name = "Burmese"
> full_name = first_name + last_name
> print(full_name)
```

# Overwriting contents of a variable

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> first_name = "Monty"
> last_name = "Python"
> full_name = first_name + last_name
> full_name
'MontyPython'
> first_name = "Burmese"
> full_name = first_name + last_name
> print(full_name)
BurmesePython
> 
```

# Commenting

- Any text following a hashtag - # - is read by Python as a comment
  - Python ignores (i.e. doesn't execute) comments

main.py



```
1  # Here is a comment – The automatic syntax highlighting in Repl.it turns it
   grey
2
3  # Now I will write some code below:
4  print("Hello, World!") # I can also place a comment at the end of a code line
5
6  # Any comments will be ignored by Python. Try this:
7  # print("Hello, Planet Earth!")
8  |
```

# DEAR FUTURE SELF,  
#  
# YOU'RE LOOKING AT THIS FILE BECAUSE  
# THE PARSE FUNCTION FINALLY BROKE.  
#  
# IT'S NOT FIXABLE. YOU HAVE TO REWRITE IT.  
# SINCERELY, PAST SELF

DEAR PAST SELF, IT'S KINDA  
CREEPY HOW YOU DO THAT.

# ALSO, IT'S PROBABLY AT LEAST  
# 2013. DID YOU EVER TAKE  
# THAT TRIP TO ICELAND?

STOP JUDGING ME!



# Commenting is:

Helping out your future  
self (and others)



Radboud Universiteit





# Readings this week

- **Think Python Book:**

Download pdf here: <https://greenteapress.com/wp/think-python-2e/>

- **Chapters 1 and 2**

- **Ask questions about readings:**

- At tutorial on Friday
- At start of lecture next Monday

# Weekly assignments

## Structure your assignment as follows:

- Write each exercise in a separate .py file, end then import each of them into main.py in replit so you can run them
- Use comments (text following a hashtag) to number subquestions (e.g. #2a:) and to answer conceptual questions in written text

## Uploading to Brightspace:

- Download all your scripts at once from replit using the "Download as zip" option, put your name in the file name
- Upload that zip file to Brightspace (under Activities → Assignments)

# Lecture 3:

# Strings, Lists,

# Dictionaries & Sets

Programming for beginners: Python

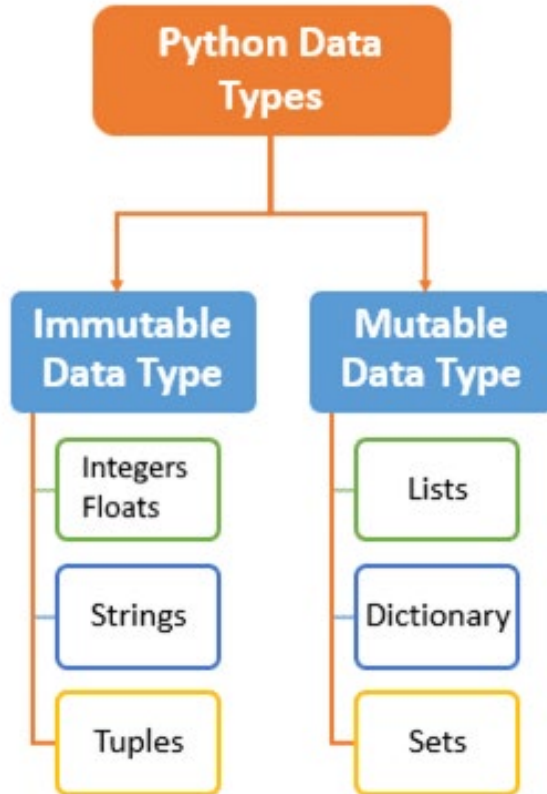
# Formal languages & Syntax

- **Python is a formal language, designed to express computations**
- **Syntax rules:**
  - **Tokens:** legal elements
    - Compare: phoneme, suffix, noun, verb, noun phrase, etc.
    - *This is @ well-structured Engli\$h sentence with invalid t\*kens in it.*
  - **Structure:** rules about how tokens are combined
    - Compare: basic word order (SOV, SVO, etc.), prefixing vs. suffixing, etc.
    - *This sentence all valid tokens has, but invalid structure with.*

# Some terminology

- **Expression:** Any combination of values, variables and operators
  - **Operator:** special symbol that represents simple computation:
    - Arithmetic operators: +, -, \*, \*\*, etc.
    - 2+2
- **Statement:** Something that can be **evaluated** by Python to give an outcome:
  - `first_name = "Monty"`
  - `print(first_name)`
- **Execution:** Python does whatever the statement tells it to do

# Python data types



# Strings

*'Hello world'*

# Indexing

- A string is a **sequence** of characters
- Can access part of a sequence by **indexing** using the **bracket operator**:
- **Important:** indexing in Python starts counting from 0!
- An index has to be an integer

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> my_string = "Monty"
```

```
> my_string[2]
```



# Indexing

- A string is a **sequence** of characters
- Can access part of a sequence by **indexing** using the **bracket operator**:
- **Important:** indexing in Python starts counting from 0!
- An index has to be an integer

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> my_string = "Monty"
```

```
> my_string[2]
```

```
'n'
```

```
> █
```

# Indexing

- You can use a variable as an index

```
> my_string = "Hello, World!"  
> my_index = 1  
> my_letter = my_string[my_index]  
> print(my_letter)
```

e

```
> 
```

# Indexing

- You can use a variable as an index

```
> my_string = "Hello, World!"  
> my_index = 1  
> my_letter = my_string[my_index]  
> print(my_letter)  
e  
> my_letter = my_string[my_index+1]  
> print(my_letter)
```

# Indexing

- You can use a variable as an index

```
> my_string = "Hello, World!"
> my_index = 1
> my_letter = my_string[my_index]
> print(my_letter)
e
> my_letter = my_string[my_index+1]
> print(my_letter)
l
> 
```

# Indexing: accessing last element

- len() is a built-in Python function that outputs the length of a string

```
> sentence = "Birds of a feather flock together"
> length = len(sentence)
> print(length)
33
> sentence[length]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
> 
```

# Indexing: accessing last element

- len() is built-in Python function that outputs the length of a string

```
> sentence = "Birds of a feather flock together"
> length = len(sentence)
> print(length)
33
> sentence[length]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
> sentence[length-1]
'r'
> 
```

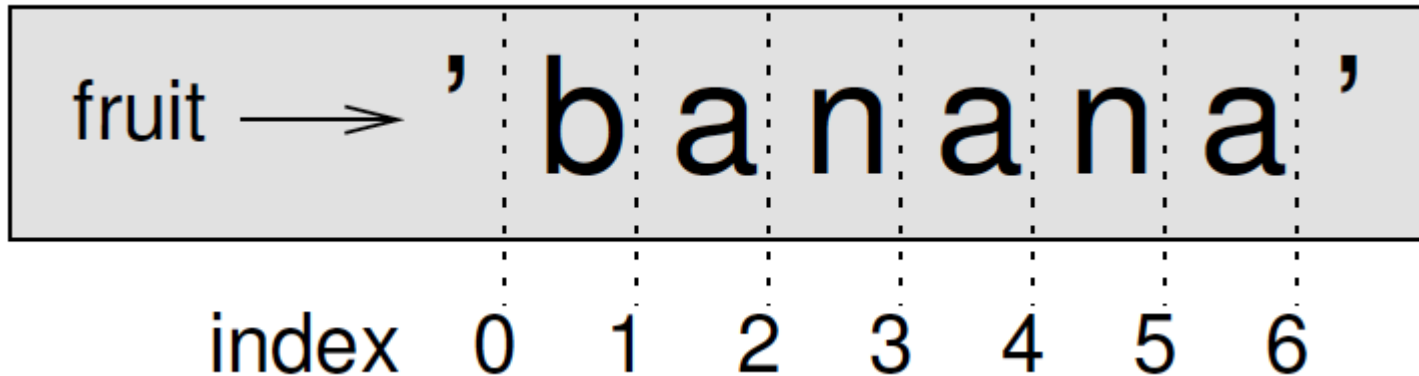
# Indexing: accessing last element

- Negative indices count backward from the end of the string

```
> sentence = "Birds of a feather flock together"  
> sentence[-1]  
'r'  
> 
```

# Slicing

- The operator **[n:m]** returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last





# Slicing

- The operator [n:m] returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last

```
> expression = "If it ain't broke, don't fix it"
> second_word = expression[3:5]
> print(second_word)
it
> 
```

# Slicing

- If you omit the **first index** (before the colon), the slice starts at the **beginning of the string**

```
> proverb = "A watched pot never boils"
> length = len(proverb)
> mid_index = int(length/2)
> first_half = proverb[:mid_index]
> print(first_half)
A watched po
> 
```

# Slicing

- If you omit the **second index**, the slice goes to the **end of the string**

```
> proverb = "A watched pot never boils"
> length = len(proverb)
> mid_index = int(length/2)
> second_half = proverb[mid_index:]
> print(second_half)
t never boils
> 
```

# Slicing

- What happens if you omit **both indices**?

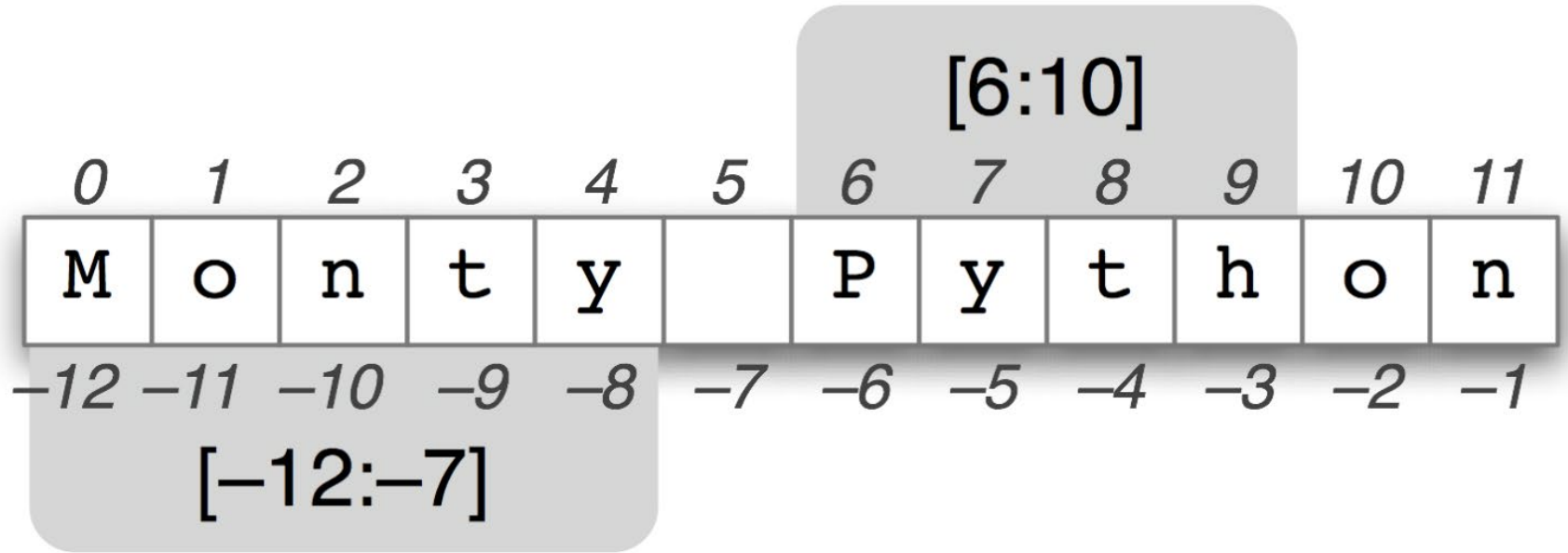
```
➤ proverb = "A watched pot never boils"  
➤ slice = proverb[:]  
➤ print(slice)
```

# Slicing

- What happens if you omit **both indices**?

```
➤ proverb = "A watched pot never boils"  
➤ slice = proverb[:]  
➤ print(slice)  
A watched pot never boils  
➤
```

# Indexing & Slicing: Summary



# String operations

## 1. Concatenation:

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> first_word = "Hello"  
> second_word = "World"  
> greeting = first_word + second_word  
> greeting  
'HelloWorld'  
> 
```

# String operations

## 2. Repetition:

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> simple_greeting = "Hello"
```

```
> rupaul_greeting = simple_greeting * 3
```

```
> rupaul_greeting  
'HelloHelloHello'
```

```
> 
```



# The in operator

- The word **in** is an operator that takes two strings and returns True if the first appears as a substring in the second, and False otherwise
- **in** is a boolean operator (i.e. returns True or False)

```
> proverb = "A watched pot never boils"  
> "pot" in proverb  
True  
> "kettle" in proverb  
False  
> |
```

# Functions vs. Methods (in short)

## Functions:

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> my_variable = "46"
> print(my_variable)
46
> type(my_variable)
<class 'str'>
> int(my_variable)
46
> float(my_variable)
46.0
> str(my_variable)
'46'
```

**Usage terminology:**  
"function call"

**Syntax:**  
output = function(argument)

## Methods:

Python 3.8.2 (default, Feb 26 2020, 02:56:10)

```
> my_string = "hELLO, wORLD!"
> my_string = my_string.lower()
> print(my_string)
hello, world!
>
> w_index = my_string.find('w')
> my_string[w_index:]
'world!'
> my_string.find('wor')
7
```

**Usage terminology:**  
"method invocation"

**Syntax:**  
output = string.method(additional arguments)

For more string methods,  
see Section 8.8 of the book

# Lists

```
my_list = ['alpha', 2, [2.3, 'hello'], {1:'b'}, 'alpha']
```

# Lists

- Like a string, a list is a **sequence of values**
- In a list, those **values can be of any type**
- The values in a list are called **elements** or sometimes **items**
- To create a list: **Enclose the elements in square brackets, and separate with commas**

```
➤ subjects = ['math', 'biology', 'english', 'physics']  
➤ grades = [7.2, 8.1, 6.5, 6.9]  
➤ group = ["4b", 2020]  
➤
```

# Lists: Indexing & Slicing

- For lists, indexing and slicing work the same way as for strings

```
> subjects = ['math', 'biology', 'english', 'physics']
> grades = [7.2, 8.1, 6.5, 6.9]
> group = ["4b", 2020]
> subjects[1]
'biology'
> grades[:3]
[7.2, 8.1, 6.5]
> group[-1]
2020
> 
```

# Nested lists

- A list can contain another list

```
> grades_per_student = [[4.3, 7.1], [8.9, 6.7], [7.5, 6.3]]  
> grades_per_student[0]  
[4.3, 7.1]  
>  
```

# Lists are mutable

- You can change elements of a list by using the bracket operator on the left-hand side of an assignment operation

```
➤ pets = ['axolotl', 'chihuahua', 'budgy']  
➤ pets[1] = 'dog'  
➤ print(pets)  
['axolotl', 'dog', 'budgy']  
➤
```



# List operations

## 1. Concatenation (same as with strings):

```
> group_a = ["Sam", "Simren", "Chris"]  
> group_b = ["Mariam", "Alex"]  
> combined = group_a + group_b  
> print(combined)  
['Sam', 'Simren', 'Chris', 'Mariam', 'Alex']  
> 
```

# List operations

## 2. Repetition (same as with strings):

```
➤ numbers = [3, 8, 5]
➤ repeated_numbers = numbers*3
➤ print(repeated_numbers)
[3, 8, 5, 3, 8, 5, 3, 8, 5]
➤
```

# The len() function & the **in** operator

- The len() function also works on lists:

```
> numbers = [3, 8, 5]
> len(numbers)
3
> 
```

- The **in** operator also works on lists:

```
> numbers = [3, 8, 5]
> my_lucky_number = 8
> my_lucky_number in numbers
True
> another_number = 4
> another_number in numbers
False
> 
```

# List methods: append

- `.append()` adds a new element to the end of a list

```
> grocery_list = ['eggs', 'apples', 'yogurt']  
> print(grocery_list)  
['eggs', 'apples', 'yogurt']  
> grocery_list.append('granola')  
> print(grocery_list)  
['eggs', 'apples', 'yogurt', 'granola']  
> █
```

For more list methods,  
see Section 10.6 of the book

# Dictionaries

```
my_dictionary = {1: [1, 2, 3, 4], 'Name': 'Geeks', 2:'Radboud'}
```

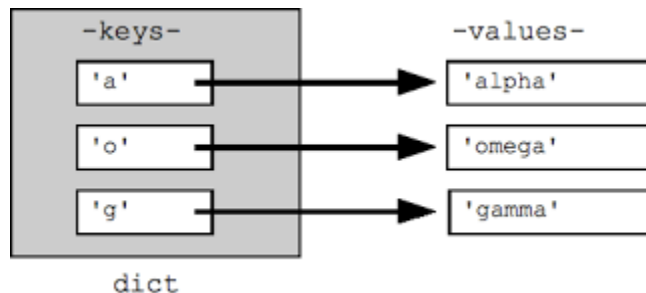
# Dictionaries

- **Difference between list and dictionary:**

- In a list, indices have to be integers
- In a dictionary, indices **can be of (almost) any type**

- Dictionary is a **mapping between:**

- A collection of **unique** indices: **keys**
- A collection of corresponding values: **values**
- The association of a key and a value is called a **key-value pair** or sometimes an item



# Dictionaries

```
> grades = {'math':7.2, 'biology':8.1, 'english':6.5, 'physics':6.9}
> english_grade = grades['english']
> print(english_grade)
6.5
>
> grades['history'] = 7.6
> print(grades)
{'math': 7.2, 'biology': 8.1, 'english': 6.5, 'physics': 6.9, 'history': 7.6}
> █
```



# Dictionary methods

- `.keys()` returns the keys in the dictionary
- `.values()` returns the values in the dictionary

```
➤ grades = {'math':7.2, 'biology':8.1, 'english':6.5, 'physics':6.9}
➤ grades.keys()
dict_keys(['math', 'biology', 'english', 'physics'])
➤ grades.values()
dict_values([7.2, 8.1, 6.5, 6.9])
➤ █
```

# The len() function & the **in** operator

- The len() function also works on dictionaries:

```
> grades = {'math':7.2, 'biology':8.1,
'english':6.5, 'physics':6.9}
> len(grades)
4
> 
```

- You can use the **in** operator on keys or values:

```
> grades = {'math':7.2, 'biology':8.1,
'english':6.5, 'physics':6.9}
> 'math' in grades.keys()
True
> 'chemistry' in grades.keys()
False
> 
```

# Quiz

I. What will be the result of the following statement?

'2'+'4'

A. 6

B. 0

C. 24

D. 8

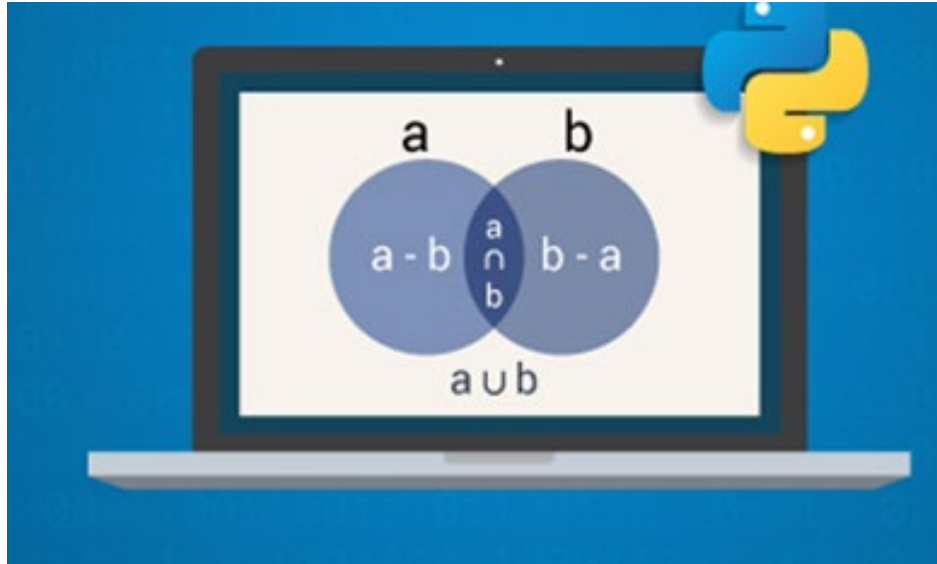
E. *Error*

# Sets

*my\_set = {1, 2, 3, 4, 5, 6, 7, 8}*

# Set References

- <https://medium.com/edureka/sets-in-python-a16b410becf4>



# Readings/Practice for this week

Radboud Universiteit



- **Think Python Book:**

Download pdf here: <https://greenteapress.com/wp/think-python-2e/>

- **Sections 8.1-8.2 + 8.4-8.5 + 8.8 (Strings)**
- **Sections 10.1-10.2 + 10.4-10.6 (Lists)**
- **Section 11.1 (Dictionaries)**
- **Section 19.5 (Sets)**

- **Strings:** <https://www.programiz.com/python-programming/string>

- **Lists:** <https://www.programiz.com/python-programming/list>

- **Dictionaries:** <https://www.programiz.com/python-programming/dictionary>

- **Sets:** <https://www.programiz.com/python-programming/set>

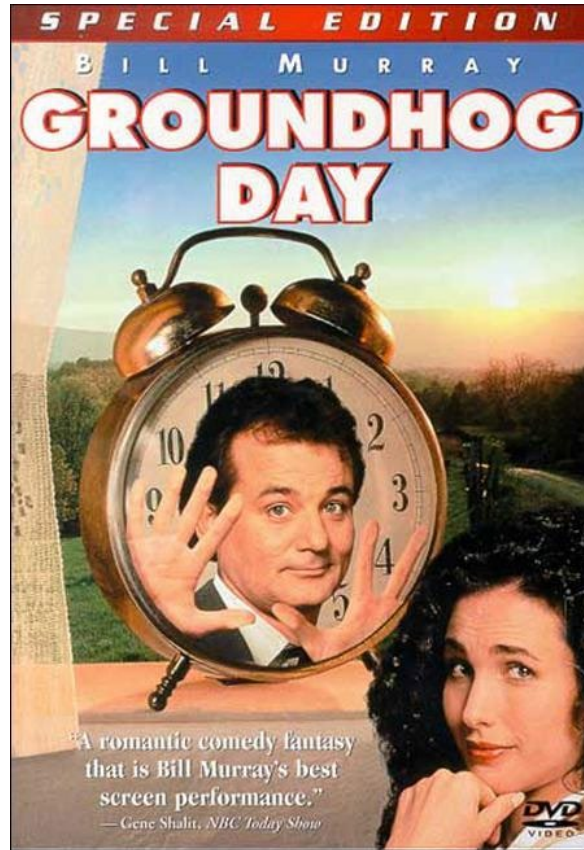
- **100% recommended Youtube channel:** [https://www.youtube.com/playlist?list=PL98qAXLA6afuh50qD2MdAj3ofYjZR\\_Phn](https://www.youtube.com/playlist?list=PL98qAXLA6afuh50qD2MdAj3ofYjZR_Phn) 44

# Lecture 4:

# Loops

Programming for beginners: Python

# Loops?



Radboud Universiteit





# Repeating the same operation with slight variation

```
1 text = "this is a very simple sentence"
2
3 text = text.replace("a", "een")
4 text = text.replace("is", "is")
5 text = text.replace("sentence", "zin")
6 text = text.replace("simple", "eenvoudige")
7 text = text.replace("this", "dit")
8 text = text.replace("very", "erg")
9
10 print(text)
```

```
dit is een erg eenvoudige zin
```

Isn't there a better way?

```
['this', 'is', 'a', 'very', 'simple', 'sentence']  
dit is een erg eenvoudige zin  
dit is een erg eenvoudige zin
```

# Iteration with a for-loop

```
1 translation = {"a":"een", "is":"is", "sentence":"zin",  
2 "simple":"eenvoudige", "this":"dit", "very":"erg"}  
3 my_sentence = "this is a very simple sentence"  
4 my_words_list = my_sentence.split()  
5 print(my_words_list)
```

```
6 # Option A  
7 for word in my_words_list:  
8     print(translation[word], end=' ')  
9  
10 print()
```

```
11  
12 # Option B  
13 for sim_en_word in translation.keys():  
14     sim_dutch_word = translation[sim_en_word]  
15     my_sentence = my_sentence.replace(sim_en_word, sim_dutch_word)  
16 print(my_sentence)
```

```
17  
18 # Of course, there are more options :)
```

Yes there is!  
Using a for-loop

for loops

# for loop with string

for\_loop.py

```
1 my_string = "Hello"
2
3 for letter in my_string:
4     print(letter)
5
6
7
```

<https://LPiPAssignment03.marieke>

H  
e  
l  
l  
o

## Syntax:

```
for character in string:
    do something with
character
```

Indent is  
meaningful  
in Python!

Is executed for  
each character  
in turn

# Iterables

- An **iterable** is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for loop
- Examples of iterables:
  - Strings } **sequences**
  - Lists }
  - Dictionaries → **collection**; iterate over **keys** or **values**
    - More information: <https://realpython.com/iterate-through-dictionary-python/>

# for loop with list: Iteration **over elements**

for\_loop.py

```
1  veggies = ['courgette', 'aubergine', 'broccoli']
2
3  for vegetable in veggies:
4      print(vegetable)
5
```

<https://LPiPAssignment03.marieke>

```
courgette
aubergine
broccoli
> []
```

## Syntax:

```
for element in iterable:
    do something with
    element
```

# for loop with list: Iteration **by index**

for\_loop.py

```
1 numbers = [2, 4, 6]
2
3 for i in range(len(numbers)):
4     numbers[i] = numbers[i]**2
5     print(numbers[i])
6
```

<https://LPiAssignment03.mariek€>

4  
16  
36  
❏

## Syntax:

```
for index in range(len(iterable)):
    do something with
iterable[index]
```

# range() function

for\_loop.py

```
1 my_range = range(3)
2
3 for number in my_range:
4     print(number)
5
```

<https://LPiPAAssignment/>

0  
1  
2

for\_loop.py

```
1 for i in range(3):
2     print("hello")
3
4
5
```

<https://LPiPAAssignment/>

hello  
hello  
hello



# Mutability

- As we have seen, **lists** are mutable: which means changes can be made *in place* to them
- Strings are immutable which means they cannot be changed in place. String methods that return a modified string create a new string

# Mutability

## Lists are mutable:

```
➤ my_list = ['banana', 'apple', 'strawberry']
➤ my_list.append('pear')
➤ my_list
['banana', 'apple', 'strawberry', 'pear']
➤
```

## Strings are immutable:

```
➤ my_string = 'banana'
➤ my_string.replace('a', 'o')
'bonono'
➤ my_string
'banana'
➤ my_string = my_string.replace('a', 'o')
➤ my_string
'bonono'
➤
```

# Dictionaries revisited

- A dictionary key has to be an immutable object (a string, a number or a tuple)
- The value that the key maps to can be of any data type

# Nested for loops

```
1  grades_per_student = [[4.3, 7.1],  
2  [8.9, 6.7], [7.5, 6.3]]  
3  avg_per_student = []  
4  for student in grades_per_student:  
5      total = 0  
6      for grade in student:  
7          total = total + grade  
8      student_avg = total/len(student)  
9      avg_per_student.append(student_avg)  
10  
11 print(avg_per_student)
```

```
[5.699999999999999, 7.800000000000001,  
6.9]  
✦ []
```

# Nested for loops

for\_loop.py

```
1  grades_per_student = [[4.3, 7.1],  
2    [8.9, 6.7], [7.5, 6.3]]  
3  avg_per_student = []  
4  for student in grades_per_student:  
5      total = 0  
6      for grade in student:  
7          total = total + grade  
8      student_avg = total/len(student)  
9      avg_per_student.append(student_avg)  
10  
11  print(avg_per_student)
```

<https://LPiPAssignment03.mariekewoe.repl.run>

```
[5.699999999999999, 7.800000000000001,  
6.9]  
[]
```

while loops

# while loops

while\_loop.py

```
1 budget = 4.50
2
3 things_i_want = {'bananas':1.99, 'yoghurt':1.69, 'lettuce':0.69,
4                  'juice':1.55, 'cookies':0.75, 'chocolate':1.50}
5
6 shopping_list = []
7 cost = 0
8 i = 0
9 while cost < budget:
10     item = list(things_i_want)[i]
11     shopping_list.append(item)
12     i = i + 1
13     cost = cost + things_i_want[item]
14
15 print(shopping_list)
```

## If True:

Run body & go  
back to top

## If False:

Exit while loop &  
continue at next  
statement

<https://LPiAssignment03.mariekewoe.repl>

```
['bananas', 'yoghurt', 'lettuce',
'juice']
```

# Beware: Infinite loops

while\_loop.py

```
1  n = 2
2
3  while n != 8:
4      n = n**2
5
6  print(n)
7
```



# Beware: Infinite loops

while\_loop.py

```
1  n = 2
```

```
2
```

```
3  while n != 8:
```

```
4      n = n**2
```

```
5
```

```
6  print(n)
```

```
7
```

Will terminate  
only when n  
exactly equals 8

# break

while\_loop.py

```
1 print("I'm blue")
2 while True:
3     print("da ba dee da ba daa")
4     response = input('> ')
5     if response == 'Please stop!':
6         break
7
8
9
10
11
12
```

```
I'm blue
da ba dee da ba daa
> haha
da ba dee da ba daa
> ok
da ba dee da ba daa
> that's enough
da ba dee da ba daa
> I said that's enough!
da ba dee da ba daa
> Please stop!
❖ █
```

Whenever you see that your  
using the same line/block of  
code repeatedly, consider  
using a loop!

# Couple of things to know for this week's readings

- < means "smaller than"
  - > means "bigger than"
  - <= means "smaller than or equal to"
  - >= means "bigger than or equal to"
  - == means "equals" (in contrast to single = for variable assignment)
  - != means "does not equal"
- If statement: Body is executed only if condition evaluates to `True`
  - Else statement: Body is executed if condition above evaluates to `False`
  - Line starting with `def function_name(argument):` function definition

→ We'll cover these over the next 2 weeks

# Readings/Tutorials for this week

- **Think Python Book:**

Download pdf here: <https://greenteapress.com/wp/think-python-2e/>

- **Section 10.3**
- **Chapter 7**
- **Section 8.3**

→ In that order!

- **Practical tutorials:**

- *While* loop: [Link](#)
- *For* loop: [Link](#)
- Extra: *break* & *continue*: [Link](#)
- Indentations: <https://www.programiz.com/python-programming/statement-indentation-comments>

Use the exercises in the  
Think Python book if you  
want more practice!

*Email us your questions or to  
request a meeting*

# Let's play!

- <https://kahoot.it>
- Laptop, smartphone, tablet...
- Your name/surname as your nick
- 10 questions
- 4 answers (4 colors)
  - Only 1 is correct
  - Only 1 attempt
- Short time (~20 seconds)
- The faster you answer the more points you get
- Good luck!



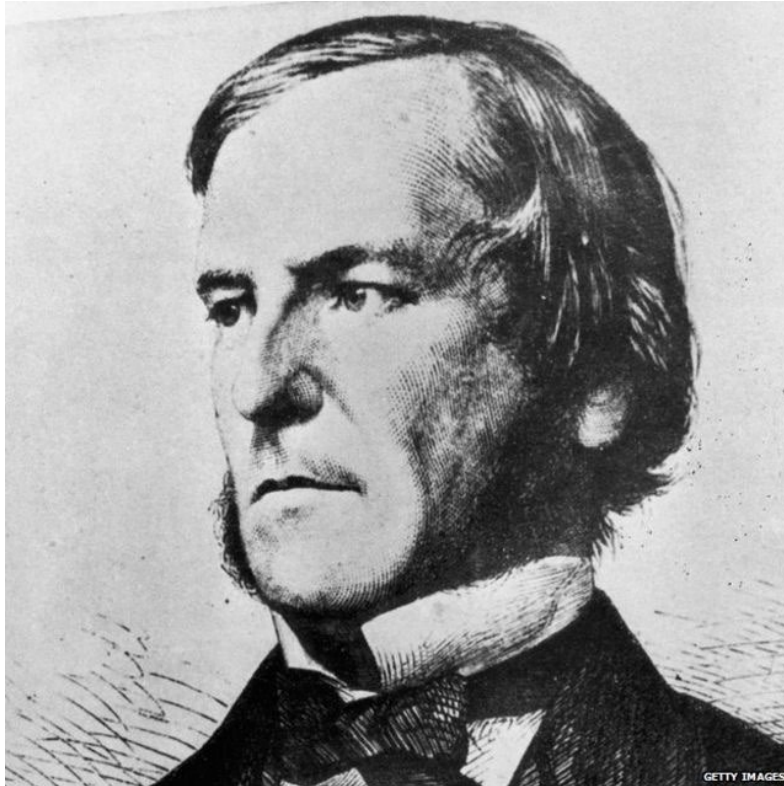
# Lecture 5:

# Conditionals

Programming for beginners: Python



# Boolean? What are you talking about?



Radboud Universiteit



# Boolean expressions

- A **Boolean expression** is an expression that evaluates to either **True** or **False**

```
> names = ['Aaliyah', 'Chris', 'Simren']  
> 'Chris' in names  
True  
> 'Jackie' in names  
False  
> 
```

# Boolean expressions

- A Boolean is its **own data type**, and has only two possible values: **True** or **False**

```
➤ type(True)
<class 'bool'>
➤
➤ type(False)
<class 'bool'>
```

# Relational operators

not to be confused with single = for variable assignment

- `x == y` # x is equal to y
- `x != y` # x is not equal to y
- `x > y` # x is greater than y
- `x < y` # x is less than y
- `x >= y` # x is greater than or equal to y
- `x <= y` # x is less than or equal to y

# Logical operators

- and
- or
- not

```
➤ 4 > 3 and 2 < 3
```

```
True
```

```
➤ 4 > 5 or 2 < 3
```

```
True
```

```
➤ names = ['Aaliyah', 'Chris', 'Simren']
```

```
➤ 'Bob' not in names
```

```
True
```

```
➤ █
```

# Logical operators

- The operands of the logical operators should be boolean expressions

**and:**

```
➤ True and True  
True  
➤ True and False  
False
```

```
➤ 4 > 3 and 2 < 3  
True
```

**or:**

```
➤ False or True  
True  
➤ False or False  
False
```

**not:**

```
➤ not True  
False  
➤ not False  
True
```

# Conditionals

# Conditional execution: The **if**-statement

main.py

```
1 object_size = 0.8
2 available_space = 1.2
3
4 if object_size < available_space:
5     print("Yay! It fits!")
```

Indent is  
meaningful  
in Python!

**Syntax:**

```
if condition:
    do
something
```

Condition has  
to be a Boolean  
expression

Is executed only if  
condition evaluates  
to True

Radboud Universiteit





# Alternative execution: **if / else**

main.py

```
1 object_size = 0.8
2 available_space = 1.2
3
4 if object_size < available_space:
5     print("Yay! It fits!")
6 else:
7     print("Ohh, bummer, it doesn't
    fit :(")
```

branches

# Chained conditionals

main.py

```
1  guests = 31
2  minimum = 20
3  maximum = 50
4
5  if guests < minimum:
6      print("Oops, too few guests!")
7  elif guests > maximum:
8      print("Oh no, too many guests!")
9  else:
10     print("Event is allowed!")
11
```

Stands for  
"else if"

# Chained conditionals & Nested conditionals

main.py

```
1 my_dict = {"monkey": "animal",
2 "banana": "fruit",
3 "bicycle": "vehicle",
4 "apple": "fruit"}
5
6 animals = []
7 fruits = []
8 vehicles = []
9
10 for item in my_dict.keys():
11     if my_dict[item] == 'animal':
12         animals.append(item)
13     elif my_dict[item] == 'fruit':
14         fruits.append(item)
15     elif my_dict[item] == 'vehicle':
16         vehicles.append(item)
```

main.py

```
1 my_dict = {"monkey": "animal",
2 "banana": "fruit",
3 "bicycle": "vehicle",
4 "apple": "fruit"}
5
6 animals = []
7 fruits = []
8 vehicles = []
9
10 for item in my_dict.keys():
11     if my_dict[item] == 'animal':
12         animals.append(item)
13     else:
14         if my_dict[item] == 'fruit':
15             fruits.append(item)
16         else:
17             vehicles.append(item)
```

Another way to do the same thing, but less readable

# Chained conditionals

main.py

```
1  x = 9
2
3  if x > 10:
4      x = x/2
5  elif x % 3 == 0:
6      x += 1
7  elif x >= 8:
8      x = x**2
```

- There is no limit to the number of `elif` statements
- If there is an `else` clause, it has to be at the end, but there doesn't have to be one

# Keep things simple



pycoders 🐍 📺  
.  
.  
.  
.  
Do Follow us @pycoders 🐍  
For More  
.  
.  
.  
#khaby #pycoders #python  
#programmer #programmerslife  
#programming #programminglife  
#programmerrepublic #worldcode  
#coding #codinglife  
#webdevelopment #webdeveloper  
#java #javascript #code #sql #perl  
#php #html #development #software  
#linux #learntocode #linuxfan #nerd  
"..."



6,787 likes

4 HOURS AGO

😊 Add a comment...

Post



```
bool result;  
  
if (value_1 == true && value_2 == true)  
    result = true;  
  
else if (value_1 == true && value_2 == false)  
    result = false;  
  
else if (value_1 == false && value_2 == true)  
    result = false;  
  
else if (value_1 == false && value_2 == false)  
    result = true;  
  
if (result == true)  
    return true;  
  
else if (result == false)  
    return false;
```

return value\_1 == value\_2

@khaby

# The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Readability counts.
- ...



Take a look before next Lecture: <https://www.python.org/dev/peps/pep-0020/>

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

# String comparison

- The relational operators also work on strings:

main.py

```
1 word = "dog"
2
3 if word == 'cat':
4     print("It's a cat")
5 else:
6     print("Not a cat")
7
```

# String comparison

- Strings are ordered alphabetically on first character

main.py

```
1 word_1 = "cat"
2 word_2 = "dog"
3
4 if word_1 < word_2:
5     print(word_1 + " comes before " + word_2)
6 elif word_2 < word_1:
7     print(word_2 + " comes before " + word_1)
8
```

<https://LPiPLecture04.mariekewoe.rep>

cat comes before dog





# String comparison

- **But beware:** All uppercase letters come before the lowercase letters in Python
- **ASCII** table: [https://miro.medium.com/max/1050/1\\*DdgD00dAdXggzMdWDt7GSA.png](https://miro.medium.com/max/1050/1*DdgD00dAdXggzMdWDt7GSA.png)

main.py

```
1 word_1 = "cat"
2 word_2 = "Dog"
3
4 if word_1 < word_2:
5     print(word_1 + " comes before " + word_2)
6 elif word_2 < word_1:
7     print(word_2 + " comes before " + word_1)
8
```

<https://LPiPLecture04.mariekewoe.repl>

Dog comes before cat

Lists revisited

# Deleting elements from a list

- Three ways:

- `list.pop(index)` # takes **index** as input, removes the element from the list and returns it
- `del list[index]` # takes **index** as input, removes the element from the list, doesn't return it
- `list.remove(element)` # takes **element** as input, and removes the first occurrence of that element

# Deleting elements from a list

`list.pop(index)`

```
❖ pets = ['cat', 'dog', 'bird']
❖ pets.pop(1)
  'dog'
❖ pets
  ['cat', 'bird']
❖
```

`del list[index]`

```
❖ pets = ['cat', 'dog', 'bird']
❖ del pets[2]
❖ pets
  ['cat', 'dog']
❖
```

```
❖ pets = ['cat', 'dog', 'bird']
❖ del pets[1:3]
❖ pets
  ['cat']
❖
```

`list.remove(element)`

```
❖ pets = ['cat', 'dog', 'bird']
❖ pets.remove('cat')
❖ pets
  ['dog', 'bird']
❖
```

```
❖ pets = ['cat', 'dog', 'cat']
❖ pets.remove('cat')
❖ pets
  ['dog', 'cat']
❖
```

# Aliasing

```
>>> list_a = [4, 5, 6]
>>> list_b = [7, 8, 9, list_a]
>>> print(list_b)
[7, 8, 9, [4, 5, 6]]
>>> list_a.append(10)
>>> print(list_b)
[7, 8, 9, [4, 5, 6, 10]]
```

- Safer to avoid aliasing when you are working with mutable objects
- Immutable objects (e.g., strings), aliasing is not as much of a problem

# Readings/Tutorials for this week

- **Think Python [Book](#):**
  - **Sections 5.1-5.7**
  - **Section 8.10**
  - **Sections 10.8-10.11 & 10.13**
- **Practical video tutorials:**
  - Booleans: [Link](#)
  - If else: [Link](#)
  - Extra - pass: [Link](#)

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian.tejedorgarcia@ru.nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Let's play!

- <https://kahoot.it>
- Laptop, smartphone, tablet...
- Your name/surname as your nick
- 10 questions
- 4 answers (4 colors)
  - Only 1 is correct
  - Only 1 attempt
- Short time (~20 seconds)
- The faster you answer the more points you get
- Good luck!





# Lecture 6:

# Functions

Programming for beginners: Python

# Functions

- A function is a named sequence of statements that performs a computation
- When you define a function, you specify the name and the sequence of statements
- Later, you can “call” the function by name, to use it on some input
- If the function does not return a value, you can say it is a 'procedure'

# Example 1: Check event size

- **In pseudocode:**

- Let "bookings" be a data structure that pairs the names of event bookings with the corresponding number of guests
- Let "minimum" be the minimum number of guests required
- Let "maximum" be the maximum number of guests allowed
- For each event booking in bookings:
  - If the number of guests is smaller than the minimum, tell the client that the number of guests is too small
  - If the number of guests is bigger than the maximum, tell the client that the number of guests is too big
  - If the number of guests is neither too big nor too small, tell the client the booking has been confirmed

# Example 1: Check event size

- **In pseudocode:**

- Let "bookings" be a **data structure that pairs** the **names** of event bookings with the **corresponding number** of guests
- Let "minimum" be the minimum number of guests required
- Let "maximum" be the maximum number of guests allowed
- **For each event booking in bookings:**
  - If the number of guests is **smaller than the minimum**, tell the client that the number of guests is too small
  - If the number of guests is **bigger than the maximum**, tell the client that the number of guests is too big
  - If the number of guests is **neither too big nor too small**, tell the client the booking has been confirmed

# Example 1: Check event size

```
1 bookings = {"wedding":54,  
2 "birthday":27,  
3 "anniversary":9}  
4 minimum = 10  
5 maximum = 30  
6  
7 for event in bookings.keys():  
8     if bookings[event] < minimum:  
9         print("The minimum number of guests for  
10            this event is "+str(minimum))  
11     elif bookings[event] > maximum:  
12         print("The number of guests exceeds the  
13            government limit imposed in relation to  
            COVID-19")  
14     else:  
15         print("Your booking has been confirmed")
```

The dictionary `.keys()` method allows the for loop to step through each of the keys in the `bookings` dictionary in turn. The variable `event` will thus first contain `"wedding"`, then `"birthday_party"`, etc.

Because the `event` variable contains a key from the dictionary at each step, we can use it to index the `bookings` dictionary, which gives us the corresponding value (the number of guests)

# Example 1: Check event size

```
1 bookings = {"wedding":54,  
2 "birthday":27,  
3 "anniversary":9}  
4 minimum = 10  
5 maximum = 30  
6  
7 for event in bookings.keys():  
8     if bookings[event] < minimum:  
9         print("The minimum number of guests for  
10            this event is "+str(minimum))  
11     elif bookings[event] > maximum:  
12         print("The number of guests exceeds the  
13            government limit imposed in relation to  
            COVID-19")  
14     else:  
15         print("Your booking has been confirmed")
```

What happens in the first step through the loop:

```
8 if bookings["wedding"] < minimum:  
9     print("The minimum number of guests for  
10        this event is "+str(minimum))  
11 elif bookings["wedding"] > maximum:  
12     print("The number of guests exceeds the  
13        government limit imposed in relation to  
14        COVID-19")
```

```
8 if 54 < minimum:  
9     print("The minimum number of guests for  
10        this event is "+str(minimum))  
11 elif 54 > maximum:  
12     print("The number of guests exceeds the  
13        government limit imposed in relation to  
14        COVID-19")
```

# Functions

- A function is a named sequence of statements that performs a computation
- When you define a function, you specify the name and the sequence of statements
- Later, you can "call" the function by name, to use it on some input

# Built-in functions

- We've already seen and used several of Python's built-in functions:
  - `print()`
  - `type()`
  - `int()`
  - `float()`
  - `str()`
  - `len()`
  - `range()`
- But what makes functions so powerful is that you can define your own



# Example 1: Check event size

```
7 def check_event_size(bookings_dict):
8     for event in bookings_dict.keys():
9         if bookings_dict[event] < minimum:
10             print("The minimum number of guests for
11                 this event is "+str(minimum))
12         elif bookings_dict[event] > maximum:
13             print("The number of guests exceeds the
14                 government limit imposed in relation to
15                 COVID-19")
16         else:
17             print("Your booking has been confirmed")
18
19 check_event_size(bookings)
```

Header of function definition

## Syntax:

```
def function_name(arguments):
    statements
```

Function call

## Example 1: Check event size

```
18 new_bookings = {"graduation":8,  
19 "award_ceremony":42,  
20 "company_outing":21}  
21  
22 check_event_size(new_bookings)
```

Now the function can be called again and again on new bookings dictionaries, and it will execute the same statements on the new input

# The `return` statement

- Functions can not only execute statements internally (as in the example above, where the function printed a given message, depending on some condition checks)
- A function can also **provide output (i.e. "return something")**

## Syntax:

```
def function_name(arguments):  
    statements  
    return output
```

# Example 2: Return event locations

- **In pseudocode:**

- Let "bookings" be a data structure that pairs the names of event bookings with the corresponding number of guests
- Let "max\_small" be the maximum number of guests that fits in the small room
- Let "maximum" be the maximum number of guests allowed
- For each event booking in bookings:
  - If the number of guests is smaller or equal to max\_small, the location should be the small room
  - If the number of guests is bigger than max\_small, but smaller than maximum, the location should be the big room
  - If the number of guests is bigger than the maximum, there is no location possible, because the event is not allowed

# Example 2: Return event locations

- **In pseudocode:**

- Let "bookings" be a **data structure that pairs** the **names** of event bookings with the **corresponding number** of guests
- Let "max\_small" be the maximum number of guests that fits in the small room
- Let "maximum" be the maximum number of guests allowed
- **For each event booking in bookings:**
  - If the number of guests is **smaller than or equal to max\_small**, the location should be the small room
  - If the number of guests is **bigger than max\_small and smaller than or equal to maximum**, the location should be the big room
  - If the number of guests is **bigger than maximum**, there is no location possible, because the event is not allowed

## Example 2: Return event locations

```
7 def find_location(bookings_dict, max_small, maximum):
8     locations = {}
9     for event in bookings_dict.keys():
10         if bookings_dict[event] <= max_small:
11             locations[event] = "Small room"
12         elif bookings_dict[event] > max_small and bookings_dict[event] <= maximum:
13             locations[event] = "Big room"
14         else:
15             locations[event] = None
16     return locations
17
18 event_locations = find_location(bookings, max_small, maximum)
```

An empty dictionary is created

At each step through the loop, the location for the current event is added as value to the locations dict

The final locations dictionary is returned

Output gets assigned to variable

# Flow of execution

- A function needs to have been defined before it can get called
- Python runs each statement in a program one at a time, in order, from top to bottom
- Function definitions do not alter the flow of execution of a program
- **However**, statements inside the function don't run until the function is called
- A function call is like a detour in the flow of execution: Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off

# What are functions good for?

- Creating a new function gives you an opportunity to name a group of statements, which makes your program **easier to read** and **debug**
- Functions can make a program smaller by **eliminating repetitive code**. Later, if you make a change, you only have to make it in one place
- You can **call a function anywhere** in your script, this makes functions more flexible than loops, with which the repetition only takes place there
- Dividing a long program into functions allows you to **debug the parts one at a time** and then assemble them into a working whole
- Well-designed functions are often useful for many programs. Once you write and debug one, you can **reuse** it on many occasions



# Readings/Tutorials for this week

- **Think Python Book:**
  - Chapter 3
  - Sections 6.1 + 6.2
- **Practical video tutorials (so useful):**
  - Functions: [Link](#)
  - *input()* function in Python: [Link](#)

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian.tejedorgarcia@ru.nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Let's play!

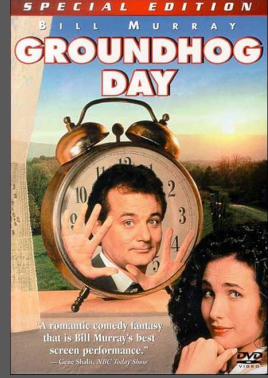
- <https://kahoot.it>
- Laptop, smartphone, tablet...
- Your name/surname as your nick
- 10 questions
- 4 answers (4 colors)
  - Only 1 is correct
  - Only 1 attempt
- Short time (~20 seconds)
- The faster you answer the more points you get
- Good luck!



# Lecture 7: Overview/Recap

Programming for beginners: Python

# Loops recap

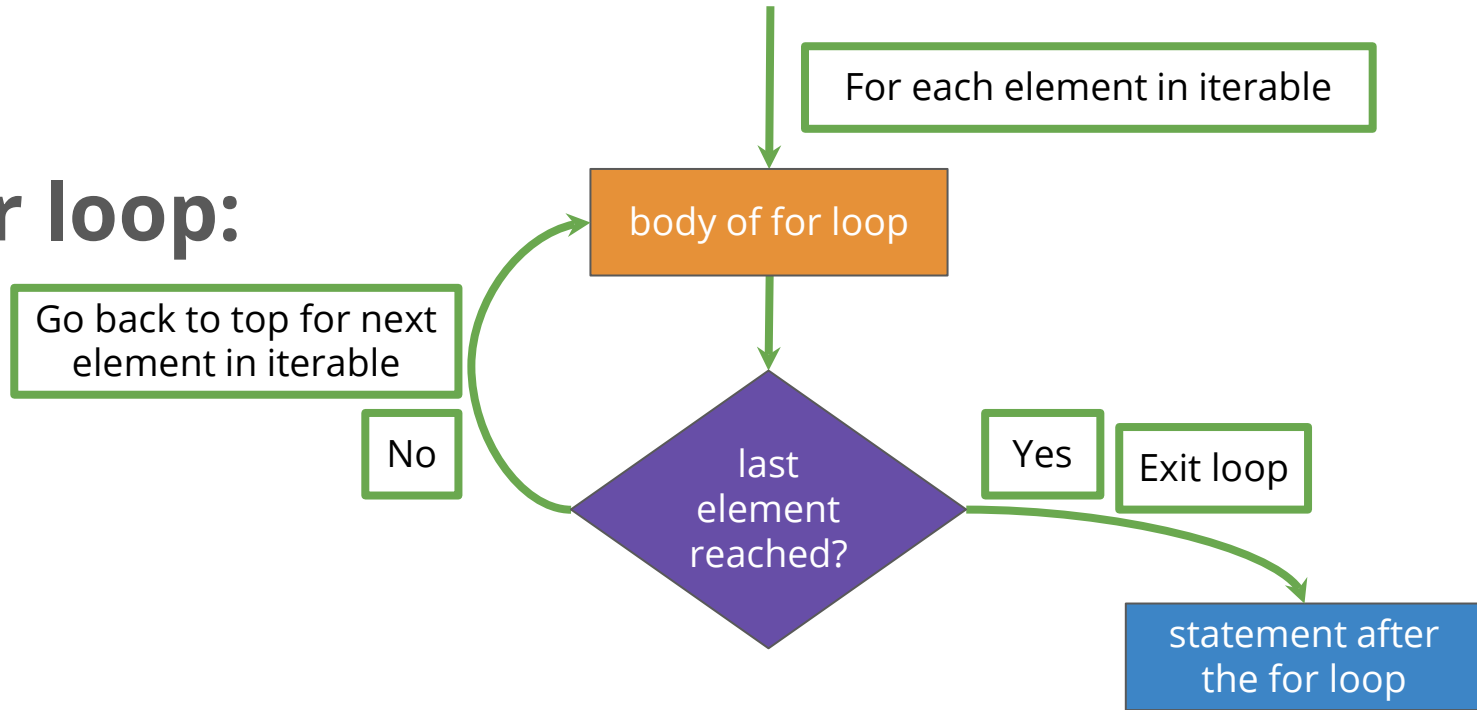


# Loops: Which one to use?

- Do you **know the number of times or number of elements** that your loop has to iterate for/over?  
→ Use a **for loop**!
- If you don't know the above, but you do **know under what condition your loop should continue/stop** running:  
→ Use a **while loop**!

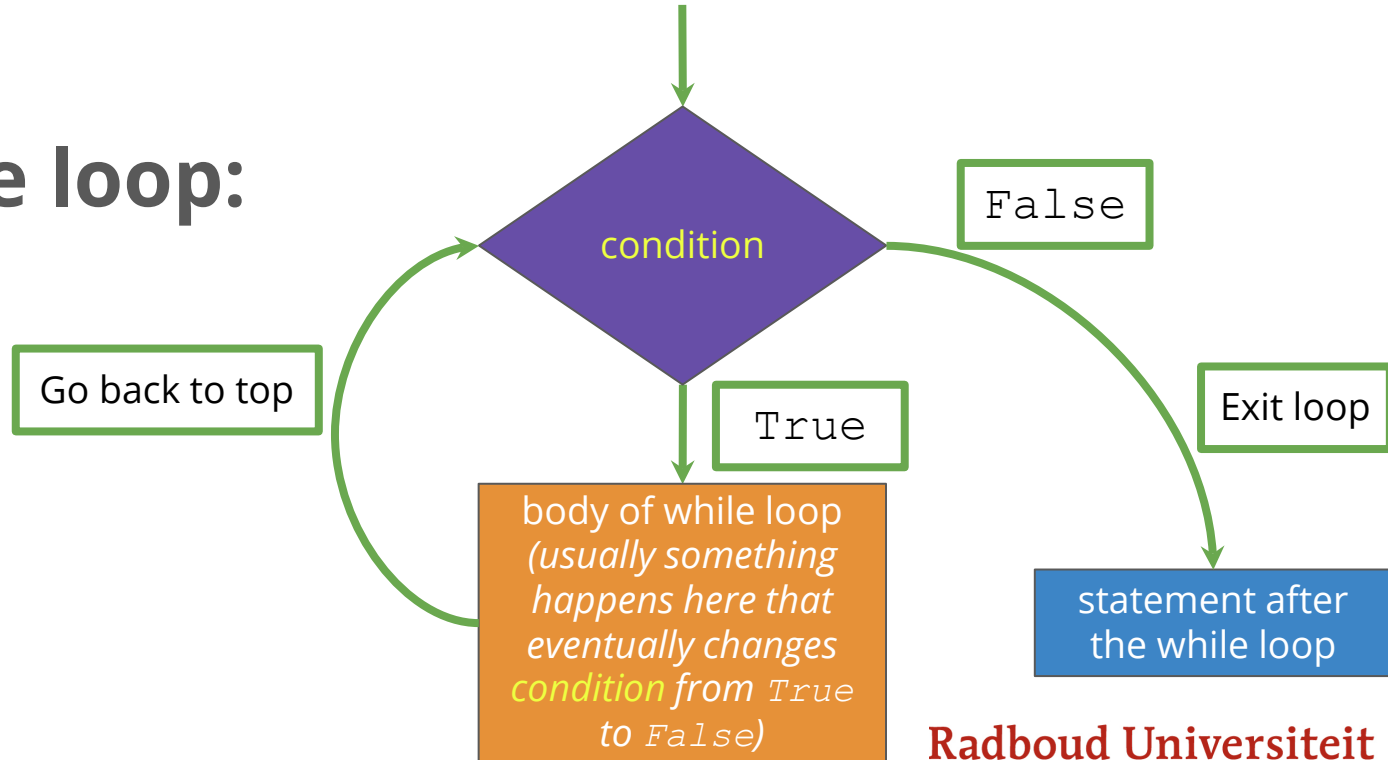
# For loop: Flow of execution

## For loop:



# While loop: Flow of execution

## While loop:





# Loops: Where to start / How to build them

## For loop

- If you only need the next **element** at each step of the loop:

```
for element in iterable:
```

At each step, the next element gets assigned to the variable `element`

- If you (also) need the **index** of the element:

```
for i in range(len(iterable)):
```

Then use `i` to index your iterable

## While loop

- The header of the while loop doesn't contain any information on what iterable to loop over, or what to call the elements in it:  
`while condition:`
- Before you can update a variable in the loop, you have to initialize it
  - Counts for dictionaries, lists, strings, etc.

# The *range* () function

- Syntax:
  - `range(start, stop, step)`
- Parameter values:
  - `start`: **Optional**. An integer number specifying at which position to start. Default is 0
  - `stop`: **Required**. An integer number specifying at which position to stop (not included)
  - `step`: **Optional**. An integer number specifying the incrementation. Default is 1

# The *range()* function

```
> for i in range(4, 10, 2):  
...     print(i)  
... 
```

# The *range()* function

```
> for i in range(4, 10, 2):  
...     print(i)  
...  
4  
6  
8  
> |
```

# What's `for i in range(len(iterable)) :` ?

- `for i in range(5):`  
    do something

→ This simply executes the statements in the body of the for loop 5 times

- `for i in range(len(iterable)) :`  
    do something

→ The for loop will step through each index of the iterable

Thereby allowing you to index each element in the iterable in turn within the body of the loop

# Common mistake: *while* + *range()*

```
while x in range(1,11):  
    print (str(x)+" cm");
```

What is missing here?

```
>>> i = 1  
  
>>> while i in range(0,10):  
...     print("Hello world", i)  
...     i = i + 1  
...  
Hello world 1  
Hello world 2  
Hello world 3  
Hello world 4  
Hello world 5  
Hello world 6  
Hello world 7  
Hello world 8  
Hello world 9  
  
>>> i  
10
```

## Common mistake: *Remove elements from a list while iterating*

[Solution](#)

```
1 list_of_num = [51, 52, 53, 54, 55, 56, 57, 58, 59]
2 for elem in list_of_num:
3     if elem == 54 or elem == 55:
4         list_of_num.remove(elem)
5 print(list_of_num)
```

[51, 52, 53, 55, 56, 57, 58, 59]

```
8 list_of_num = [51, 52, 53, 54, 55, 56, 57, 58, 59]
9 for i in range(len(list_of_num)):
10    if list_of_num[i] == 54 or list_of_num[i] == 55:
11        list_of_num.remove(list_of_num[i])
12 print(list_of_num)
```

**IndexError: list index out of range**

# Functions recap



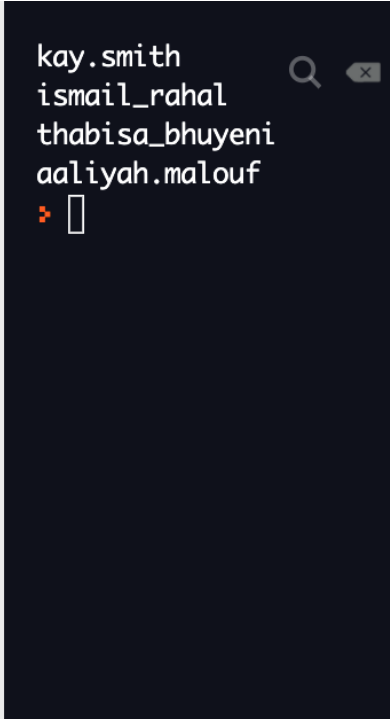
# Functions: What do they do & What are they for

- Functions allow you to encapsulate a group of statements (which together make up an operation that stands on its own to some extent) under a meaningful name  
→ which then allows you to reuse that function anywhere else in your script

# Functions: What do they do & What they're for

```
1 email_addresses = ["kay.smith@hotmail.com",
2 "ismail_rahall@let.ru.nl",
3 "thabisa_bhuyeni@gmail.com",
4 "aaliyah.malouf@gmail.com"]
5
6 def name_extractor(email):
7     at_index = email.find("@")
8     name = email[:at_index]
9     return name
10
11 name_dict = {}
12 for email in email_addresses:
13     name = name_extractor(email)
14     name_dict[name] = email
15
16 for name in name_dict.keys():
17     print(name)
```

Function call



```
kay.smith
ismail_rahall
thabisa_bhuyeni
aaliyah.malouf
> []
```

# Local vs. global variables (I)

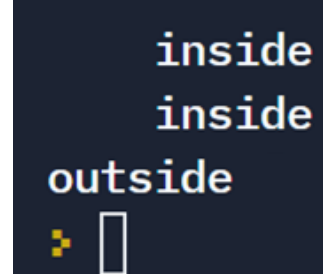
```
7 def find_location(bookings_dict, max_small, maximum):
8     locations = {}
9     for event in bookings_dict.keys():
10         if bookings_dict[event] <= max_small:
11             locations[event] = "Small room"
12         elif bookings_dict[event] > max_small and bookings_dict
13             [event] <= maximum:
14             locations[event] = "Big room"
15         else:
16             locations[event] = None
17     return locations
18
19 event_locations = find_location(bookings, max_small, maximum)
20 print(locations)
```

This throws an error, because variables that are created inside a function, only exist within that function

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    import return_event_location
  File "/home/runner/LPiPLecture05Function
s/return_event_location.py", line 20, in <
module>
    print(locations)
NameError: name 'locations' is not defined
```

# Local vs. global variables (II)

```
1  my_list = [1, 2, 3, 4]
2
3  def m():
4      print("\tinside", my_list)
5      my_list[0]=1000
6      my_list.append(10000)
7      print("\tinside", my_list)
8      return None
9
10 m()
11 print("outside", my_list)
```



```
    inside
    inside
outside
> |
```

# Local vs. global variables (II)

```
1  my_list = [1, 2, 3, 4]
2
3  def m():
4      print("\tinside", my_list)
5      my_list[0]=1000
6      my_list.append(10000)
7      print("\tinside", my_list)
8      return None
9
10 m()
11 print("outside", my_list)
```

```
inside [1, 2, 3, 4]
inside [1000, 2, 3, 4, 10000]
outside [1000, 2, 3, 4, 10000]
> []
```

# Local vs. global variables (III)

```
1  my_list = [1, 2, 3, 4]
2  my_num = 18
3
4  def m():
5      my_list = ["A", "B", "C"] # new var my_list, local
6      my_list = my_list + [1]
7      my_num = 1000 # new var my_num, local
8      my_num = my_num + 1000
9      print("\tinside", my_list)
10     print("\tinside", my_num)
11     return None
12
13 m()
14
15 print("outside", my_list)
16 print("outside", my_num)
```

inside  
inside  
outside  
outside

# Local vs. global variables (III)

```
1  my_list = [1, 2, 3, 4]
2  my_num = 18
3
4  def m():
5      my_list = ["A", "B", "C"] # new var my_list, local
6      my_list = my_list + [1]
7      my_num = 1000 # new var my_num, local
8      my_num = my_num + 1000
9      print("\tinside", my_list)
10     print("\tinside", my_num)
11     return None
12
13 m()
14
15 print("outside", my_list)
16 print("outside", my_num)
```

```
inside ['A', 'B', 'C', 1]
inside 2000
outside [1, 2, 3, 4]
outside 18
> 
```

# Bracket types



# Different bracket types: when to use [], {} and ()

## Defining variables of different data types:

- **[ ]** (=square brackets) for list

- Example:

```
my_list = [2, 5, 3]
```

- **{ }** (=curly brackets) for dictionary

- Example:

```
my_dictionary = {"Robin":29,  
                 "Kate":24,  
                 "Sam":26}
```

# Different bracket types: when to use [], {} and ()

Indexing & Slicing (of a list, string, or, in case of a dictionary, using the `.keys()`, `.values()`, or `.items()` method):

- `[]` (=square brackets) for all data types (so indexing and slicing *always* use square brackets, no matter what the data type is)
  - Examples:
    - `first_character = my_string[0]`
    - `first_slice = my_list[0:5]`
    - `first_key = my_dictionary.keys()[0]`
    - `first_value = my_dictionary.values()[0]`
    - `first_key, first_value = my_dictionary.items()[0]`

# Different bracket types: when to use [], {} and ()

## Input arguments of a function or method

- **()** (=round brackets or parentheses)
  - Examples:
    - Function definition:

```
def my_function(input_arg_1, input_arg_2):
```
    - Function call:

```
my_functions_output = my_function(input_1, input_2)
```
    - Method call:
      - `my_string.find("@")`
      - `my_list.append(35)`

# Nested indexing

# Nested indexing

- To index an object that's inside another object, simply string the indexing brackets (i.e. square brackets) together one after the other, starting with the "highest-level" object:

```
> nested_list = [[2, 5], [7, 4], [3, 1]]  
> nested_list[1]  
[7, 4]  
> nested_list[1][0]
```

# Nested indexing

```
> nested_list = [[2, 5], [7, 4], [3, 1]]  
> nested_list[1]  
[7, 4]  
> nested_list[1][0]  
7  
> 
```

# Indentation

# Indentation

- It's important to think about what should be inside and outside of a body of a conditional statement, loop, or function

```
1 numbers_a = [1.7, 5.3, 9.4, 3.9]
2 numbers_b = [4.3, 8.6, 3.7, 7.2]
3
4 def multiply_elementwise(list_a, list_b):
5     product_list = []
6     for i in range(len(list_a)):
7         product = list_a[i] * list_b[i]
8         product_list.append(product)
9     return product_list
```

```
1 numbers_a = [1.7, 5.3, 9.4, 3.9]
2 numbers_b = [4.3, 8.6, 3.7, 7.2]
3
4 def multiply_elementwise(list_a, list_b):
5     product_list = []
6     for i in range(len(list_a)):
7         product = list_a[i] * list_b[i]
8         product_list.append(product)
9     return product_list
```



# Mid-term assignment

- **Available:** Will be released **Friday** (the 22th) morning at the latest
- **Form:** Will give you an idea of what the take-home exam at the end of the course will look like.  
But also: will not look very different from the weekly assignment you've had so far
- **Deadline: Wednesday 3th of November at 9:00** (9 AM) CEST
- **Feedback:** The mid-term assignment will not count towards your grade, but you will get in-depth, individual feedback -> **Don't copy other's assig.**
- **Don't use previous years' solutions.**
- **Wednesday:** In Friday's tutorial you can start working on the mid-term assignment; we will answer your questions to help you along

# Readings this week

- No new readings! Just revise the chapters and sections from the Think Python book that we've read so far:
  - **Chapter 1**
  - **Chapter 2**
  - **Chapter 3**
  - **Chapter 5**
  - **Sections 6.1+6.2**
  - **Chapter 7**
  - **Chapter 8**
  - **Chapter 10**
  - **Section 11.1**
- Don't forget to practice at home with this YT channel: [Link](#)

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian . tejedorgarcia @ ru . nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Let's play!

- <https://kahoot.it>
- Laptop, smartphone, tablet...
- Your name/surname as your nick
- 10 questions
- 4 answers (4 colors)
  - Only 1 is correct
  - Only 1 attempt
- Short time (~20 seconds)
- The faster you answer the more points you get
- Good luck!



# Lecture 8:

# Problem Solving

Programming for Beginners: Python

Radboud Universiteit



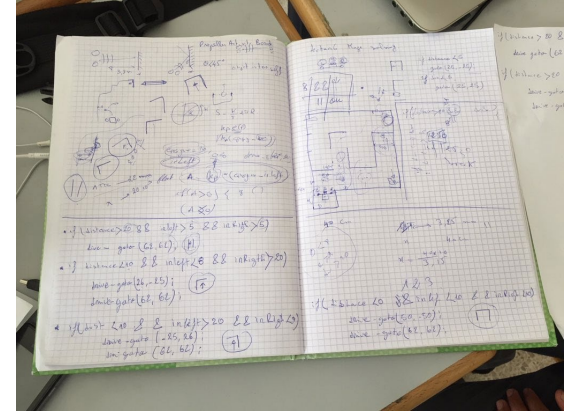
# Problem Solving

- Understand
  - Read the exercise
    - read it again
      - and again
  - Try to get the essence and ignore the redundancy
  - Rephrase it
    - explain to a(n imaginary) child (or a [rubber duck](#))
  - What goes in?
  - What goes out?



# Problem Solving

- Break it down
  - Take pen and paper
  - Write down the different (larger steps)
    - break down the large steps into smaller steps
      - break down the smaller steps into tinier steps
        - etc.
  - If you don't see the Python solution yet, stick with pseudo coding or even plain language
  - Simplify the problem if necessary
    - e.g. you have to return in alphabetical order but do not know how to do, leave it
  - Remember: often there are many solutions



# Problem Solving

- When you are stuck
  - Do not panic (for too long)
  - Debug
    - go step by step through your code
    - try to find where you make a (thinking/programming) error
    - print to screen and compare with what you expect
      - if you do not have an expectation, it is too complicated
  - Take a step back
    - is your initial solution ok
    - can you take a different approach
  - If necessary: start all over again
  - Google
    - if you know what to ask, it can be very useful
    - understand what you find, don't just implement something



# Problem Solving

- Practice

- e.g. <https://coderbyte.com/challenges>

## Longest Word

Not Completed | Easy | Solutions: 158654

[View all user solutions ▶](#)[Discussion](#)

easy

[Solutions](#)[Discussion](#)

### Longest Word

Have the function `LongestWord(sen)` take the `sen` parameter being passed and return the longest word in the string. If there are two or more words that are the same length, return the first word from the string with that length. Ignore punctuation and assume `sen` will not be empty. Words may also contain numbers, for example "Hello world123 567"

### Examples

Input: "fun&!! time"  
Output: time

Input: "I love dogs"  
Output: love

Python3

Vim Emacs

Light Theme

Reset Code

```
1 def LongestWord(sen):
2
3     # code goes here
4     return sen
5
6     # keep this function call here
7     print(LongestWord(input()))
```

Run Code

Run Test Cases

Submit

"fun&amp;!! time"

Auto-clear Clear Log

output logs will appear here

# Breaking it down (1): have a meal tonight

- How?
  - **Cook it yourself**
  - Take away
  - Delivery



# Breaking it down: **cook** a meal tonight

1. Find a recipe
2. Make a list of ingredients you need to buy
3. Go to shop to buy ingredients
4. Do the cooking

# Breaking it down: cook a meal tonight

## 1. Find a recipe

- Get a couple of cookbooks (or go on the internet)
- Look for a recipe you and your table companions like
- Check how difficult it is to realise

2. Make a list of ingredients you need to buy

3. Go to shop to buy ingredients

4. Do the cooking

# Breaking it down: cook a meal tonight

1. Find a recipe
- 2. Make a list of ingredients you need to buy**
  - Check which ingredients you already have
  - Write down the ingredients that are missing
  - Calculate how much you need from everything based upon recipe and number of eaters
3. Go to shop to buy ingredients
4. Do the cooking

# Breaking it down: cook a meal tonight

1. Find a recipe
2. Make a list of ingredients you need to buy
- 3. Go to shop to buy ingredients**
  - Make sure you have a payment method
  - Take the car/bike/bus
  - Go to the shop
  - Get the ingredients
  - Pay
  - Go back home
4. Do the cooking

# Breaking it down: cook a meal tonight

1. Find a recipe
2. Make a list of ingredients you need to buy
3. Go to shop to buy ingredients
- 4. Do the cooking**
  - Prepare the ingredients
  - Get the pans
  - Put on the stove
  - Follow the recipe

# Breaking it down: cook a meal tonight

- In this real life example you can break it down into the smallest detail
- In programming you can stop when you can translate it to Python
- Let's see now a real practical example...



## Breaking it down (2): throwing the dice



- Write a script that contains a function that returns a list of the length of the argument with which the function is called and that contains the throws of a dice (random number from 1 through 6). After the function call, print the mean of all throws and the distribution (i.e. how many 1s, 2s, 3s, 4s, 5s, 6s). You must use modules random, statistics and collections to get the random number, to compute the mean and to get the distribution.

Thus the function call looks like this

```
dice_throws = throw_dice(100000)
```

where dice\_throws is a list with 100,000 random numbers between 1 and 6.

Then you print the (floating point) mean of all the numbers in that list and how often the numbers 1-6 appear in that list.

## Read again

- Write a script that contains a function that returns a list of the length of the argument with which the function is called and that contains the throws of a dice (random number from 1 through 6). After the function call, print the mean of all throws and the distribution (i.e. how many 1s, 2s, 3s, 4s, 5s, 6s). You must use modules random, statistics and collections to get the random number, to compute the mean and to get the distribution.

Thus the function call looks like this

```
dice_throws = throw_dice(100000)
```

where dice\_throws is a list with 100,000 random numbers between 1 and 6.

Then you print the (floating point) mean of all the numbers in that list and how often the numbers 1-6 appear in that list.

# Getting the **essence**

- Write a script that contains a **function** that **returns** a **list** of the length of the argument with which the function is called and that **contains** the **throws of a dice** (random number from 1 through 6). After the function call, **print** the **mean** of **all throws** and the **distribution** (i.e. how many 1s, 2s, 3s, 4s, 5s, 6s). You must use modules random, statistics and collections to get the random number, to compute the mean and to get the distribution.

Thus the function call looks like this

```
dice_throws = throw_dice(100000)
```

where dice\_throws is a list with 100,000 random numbers between 1 and 6.

Then you **print** the (floating point) **mean** of all the numbers in that list and **how often the numbers 1-6 appear** in that list.

# Rephrasing

- I have to write a script with a function that returns a list with dice rolls and then print the mean and distribution of that list.

# Input / Output

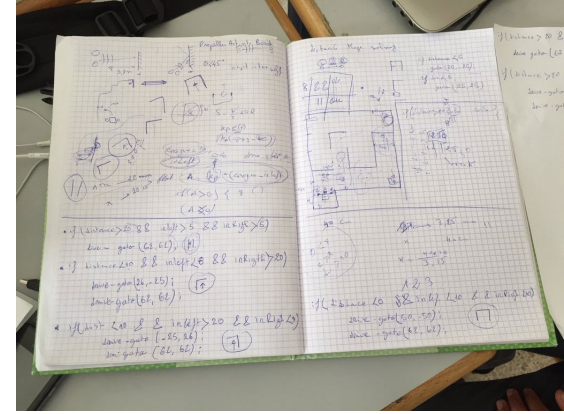
- Input of the function: number of dice rolls
  - Output of the function: list with dice rolls (with length of input)
  - Output of the script (print): mean and distribution of (list with) dice rolls
- 
- Write a script that contains a **function** that **returns** a **list** of the length of the argument with which the function is called and that **contains** the **throws of a dice** (random number from 1 through 6). After the function call, **print** the **mean** of **all throws** and the **distribution** (i.e. how many 1s, 2s, 3s, 4s, 5s, 6s). You must use modules random, statistics and collections to get the random number, to compute the mean and to get the distribution.  
Thus the function call looks like this  

```
dice_throws = throw_dice(100000)
```

  
where dice\_throws is a list with 100,000 random numbers between 1 and 6.  
Then you print the (floating point) mean of all the numbers in that list and how often the numbers 1-6 appear in that list.

# Pen and paper

- function with parameter number of dice rolls
- roll the dice as many times as in the parameter
- store the result somewhere
- return all results of dice rolls
- print mean and distribution



# Breaking it down: throwing the dice

- function with parameter number of dice rolls
  - `def throw_dice(number_of_dice_rolls)`
- roll the dice as many times as in the parameter
  - **loop? for? while?**
- store the result somewhere
  - **list? dictionary? something else?**
- return all results of dice rolls
  - **return dice\_rolls**
- print mean and distribution
  - compute mean
  - compute distribution
  - print

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- roll the dice as many times as in the parameter
  - loop? for? while?
    - **number\_of\_dice\_rolls times**
- store the result somewhere
  - **list**
- return all results of dice rolls
  - **return** `dice_rolls`
- print mean and distribution
  - compute mean
    - **go back to exercise text**
  - compute distribution
  - print



# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- roll the dice as many times as in the parameter
  - loop? for? while?
    - `number_of_dice_rolls` times
      - **range!**
- store the result in a list
  - **append**
- `return dice_rolls`
- print mean and distribution
  - compute mean
    - go back to exercise text
  - compute distribution
  - print

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- roll the dice as many times as in the parameter
  - **`for i in range(number_of_dice_rolls)`**
  - **roll a dice**
- store the result in a list
  - `append`
- `return dice_rolls`
- print mean and distribution
  - compute mean
    - go back to exercise text
  - compute distribution
  - print

# Stuck!

- `def throw_dice(number_of_dice_rolls)`
- roll the dice as many times as in the parameter
  - `for i in range(number_of_dice_rolls)`
  - **roll a dice?**

# Stuck => Google!

- `def throw_dice(number_of_dice_rolls)`
- roll the dice as many times as in the parameter
  - `for i in range(number_of_dice_rolls)`
  - roll a dice



python roll a dice



Remember: Python 3

Ongeveer 7.900.000 resultaten (0,43 seconden)

 [www.pythonforbeginners.com](#) > ga...  [Vertaal deze pagina](#)

## Python Game : Rolling the dice - PythonForBeginners.com

1 okt. 2012 — `import random min = 1 max = 6 roll_again = "yes" while roll_again == "yes" or roll_again == "y": print "Rolling the dices..." print "The values are..." print random. randint(min, max) print random. randint(min, max) roll_again = raw_input("Roll the dices again?")`

 [stackoverflow.com](#) > questions > dic...  [Vertaal deze pagina](#)

## Dice rolling simulator in Python - Stack Overflow

17 mei 2017 — Let's walk through the process: You already know what you need to generate random numbers. `import random` (or you could be more specific ...

6 antwoorden

simulating <b>rolling 2 dice</b> in <b>Python</b> - Stack Overflow	1 antwoord	11 okt. 2015
<b>dice roll</b> simulation in <b>python</b> - Stack Overflow	5 antwoorden	6 okt. 2018
Creating <b>dice</b> simulator in <b>Python</b> , how to find ...	1 antwoord	4 sep. 2018
Random <b>Dice Roll</b> Game in <b>Python</b> - Stack Overflow	1 antwoord	22 aug. 2016

[Meer resultaten van stackoverflow.com](#)

 [datascienceunlimited.tech](#) > step-by-...  [Vertaal deze pagina](#)

## Step by Step: Coding a Dice Roll Simulator in Python ...

Introduction. Step 1: Print Random Number between 1 and 6. Running the program for the first time. Step 2: Ask the User For Another **Dice Roll**. Our code thus far: Step 3: Ask User for Maximum and Minimum Die Value. Step 4: Improving How We Deal with Input. The Final Code for the Command-Line Calculator. Conclusion.

# Python for beginners

```
1  import random
2  min = 1
3  max = 6
4
5  roll_again = "yes"
6
7  while roll_again == "yes" or roll_again == "y":
8      print("Rolling the dices...")
9      print("The values are....")
10     print(random.randint(min, max))
11     print(random.randint(min, max))
12
13     roll_again = input("Roll the dices again?")
```

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- `for i in range(number_of_dice_rolls)`
  - **`dice_roll = random.randint(1,6)`**
- store the result in a list
  - `append`
- `return dice_rolls`
- print mean and distribution
  - compute mean
    - go back to exercise text
  - compute distribution
  - print

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- `for i in range(number_of_dice_rolls)`
  - `dice_roll = random.randint(1,6)`
  - **`dice_rolls.append(dice_roll)`**
- `return dice_rolls`
- print mean and distribution
  - compute mean
    - go back to exercise text
  - compute distribution
  - print



# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- `for i in range(number_of_dice_rolls)`
  - `dice_roll = random.randint(1,6)`
  - `dice_rolls.append(dice_roll)`
- `return dice_rolls`
- **print mean and distribution**
  - compute mean
    - go back to exercise text
    - ***You must use modules random, statistics and collections to get the random number, to compute the mean and to get the distribution.***
  - compute distribution
  - print

# Google!



python mean



Alle

Afbeeldingen

Video's

Nieuws

Boeken

Meer

Instellingen

Tools

Ongeveer 304.000.000 resultaten (0,53 seconden)

[www.geeksforgeeks.org > python-st...](https://www.geeksforgeeks.org/python-statistics-mean-function/) Vertaal deze pagina

## Python statistics | mean() function - GeeksforGeeks

13 apr. 2018 — **Python** statistics | **mean()** function **mean()** function can be used to calculate **mean/average** of a given list of numbers. It returns **mean** of the data set passed as parameters. Arithmetic **mean** is the sum of data divided by the number of data-points.

# GeeksforGeeks

```
# Importing the statistics module
import statistics

# list of positive integer numbers
data1 = [1, 3, 4, 5, 7, 9, 2]

x = statistics.mean(data1)
```

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- `for i in range(number_of_dice_rolls)`
  - `dice_roll = random.randint(1,6)`
  - `dice_rolls.append(dice_roll)`
- `return dice_rolls`
- **print mean and distribution**
  - `dice_rolls_mean = statistics.mean(dice_throws)`
  - compute distribution
  - print

# Breaking it down: throwing the dice

- compute distribution
  - *You must use modules **random**, **statistics** and **collections** to get the random number, to compute the mean and to get the distribution.*

from <https://docs.python.org/3/library/collections.html>

```
class collections.Counter([iterable-or-mapping])
```

A `Counter` is a `dict` subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The `Counter` class is similar to bags or multisets in other languages.

```
{number1: n_times, number2: n_times, ...}
```

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- `for i in range(number_of_dice_rolls)`
  - `dice_roll = random.randint(1,6)`
  - `dice_rolls.append(dice_roll)`
- `return dice_rolls`
- `print mean and distribution`
  - `dice_rolls_mean = statistics.mean(dice_throws)`
  - **`dice_rolls_distribution = collections.Counter(dice_throws)`**
  - **`print`**

# Breaking it down: throwing the dice

- `def throw_dice(number_of_dice_rolls)`
- `for i in range(number_of_dice_rolls)`
  - `dice_roll = random.randint(1,6)`
  - `dice_rolls.append(dice_roll)`
- `return dice_rolls`
- `dice_rolls_mean = statistics.mean(dice_throws)`
- `dice_rolls_distribution = collections.Counter(dice_throws)`
- **`print(dice_rolls_mean,dice_rolls_distribution)`**

# Breaking it down: throwing the dice



```
1 import random, statistics, collections
2
3 def throw_dice(number_of_dice_rolls):
4     dice_rolls = []
5     for i in range(number_of_dice_rolls):
6         dice_roll = random.randint(1,6)
7         dice_rolls.append(dice_roll)
8     return dice_rolls
9
10 dice_throws = throw_dice(100000)
11 dice_rolls_mean = statistics.mean(dice_throws)
12 dice_rolls_distribution = collections.Counter(dice_throws)
13
14 print(dice_rolls_mean, dice_rolls_distribution)
```

```
3.50412 Counter({6: 16907, 1: 16771, 4: 16672, 2: 16627,
5: 16594, 3: 16429})
```



# Practice for this week

- **coderbyte.com**
  - <https://coderbyte.com/> (View Challenges without the lock (unless you have a paid subscription))
- **Assignments 1-6**
  - Try again those exercises you found difficulties and send them to Cristian or Eric to obtain some feedback.

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian.tejedorgarcia@ru.nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Lecture 9:

# File I/O

Programming for Beginners: Python

# Files

- Files can be **read** from or **written** to
- Files are connected through a **file handle**
- In this course only **text** files are treated (in contrast to **binary** files)
- Syntax:**  
`file_object = open(file_name, <<mode>>) # mode: r, w, a (and others)`

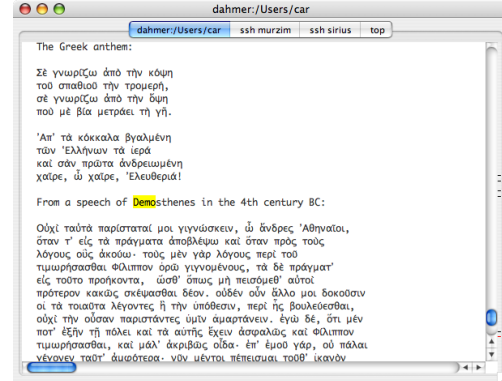
## Examples:

```
input_file = open('source.txt')
```

mode='rt' is default

```
output_file = open('target.txt', mode='w')
```

```
output_file = open('target.txt', 'w')
```



# Files

```
input_file = open('source.txt', 'r')  
text = input_file.read()           # read the whole file at once (bad  
performance)
```

```
line = input_file.readline()       # read one line at the time (not useful)
```

BUT most of the times you use:

```
for line in input_file:             # read one line at the time  
    do something                    # and do something
```

```
output_file = open('target.txt', 'w')  
output_file.write(string)           # write always a string to output_file
```

# Example: Translate file using lexicon

- **In pseudocode:**

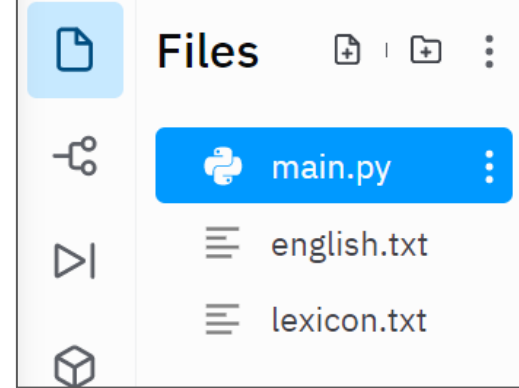
- Read lexicon
- Read the English text file
- Translate line by line
- Write translation to new file

lexicon.txt

```
1 a een
2 file bestand
3 is is
4 short kort
5 this dit
6 very zeer
```

english.txt

```
1 this is a
2 very short file
```



# Example: Create function that **reads lexicon file**

- **In pseudocode:**

- Let "lexicon" be an empty data structure that pairs the English words with corresponding Dutch words
- Open the lexicon file
- For each line in the lexicon file:
  - **Remove the newline character**
  - Split the line in two words, English and Dutch
  - Store the English and the Dutch word in the lexicon
- Return the lexicon

lexicon.txt

```
1 a een
2 file bestand
3 is is
4 short kort
5 this dit
6 very zeer
```

# Example: Create function that **reads lexicon file**

- **In pseudocode:**

- Let "lexicon" be an empty **data structure** that **pairs** the **English words** with corresponding **Dutch words**
- **Open** the lexicon **file**
- **For each line** in the **lexicon file**:
  - **Remove** the newline character
  - **Split** the line in two words, English and Dutch
  - **Store** the English and the Dutch word **in the lexicon**
- **Return** the lexicon

lexicon.txt

```
1 a een
2 file bestand
3 is is
4 short kort
5 this dit
6 very zeer
```



# Example: Create function that reads lexicon file

```
1 def read_lexicon(lexicon_file_name):  
2     lexicon = {}  
3     lexicon_file = open(lexicon_file_name)  
4     for line in lexicon_file:  
5         line = line.strip()  
6         english,dutch = line.split()  
7         lexicon[english] = dutch  
8     return(lexicon)  
9     lexicon_file.close()  
10  
11 lexicon = read_lexicon('lexicon.txt')
```

The `open` function creates a file handle named `lexicon_file` that points to the file with the name that is stored in the variable `lexicon_file_name`.

The `for` loop reads each line of the file one by one.

The `close` method closes the file handle. Don't forget it!

This is a [tuple](#) (similar to a list). We know beforehand `split()` will return two values always.

# Example: Translate file and write to new file

- In pseudocode:

- Call the function that reads the lexicon
- Open the file with the English text for reading (source file)
- Open the file to write the Dutch text (target file)
- For each line in the source file:
  - Remove the newline character and split the line in words
  - For each word in the line:
    - Look up in the lexicon and append translated word to list of new words
  - Make a string of the list of new words
  - Write the string to the target file

lexicon.txt

```
1 a een
2 file bestand
3 is is
4 short kort
5 this dit
6 very zeer
```

english.txt

```
1 this is a
2 very short file
```

Radboud Universiteit



# Example: Translate file and write to new file

- In pseudocode:

- Call the **function** that reads the lexicon
- **Open the file** with the English text for **reading** (source file)
- **Open the file** to **write** the Dutch text (target file)
- **For each line** in the **source file**:
  - **Remove** the newline character and **split** the line in words
  - **For each word** in the **line**:
    - **Look up** in the lexicon and **append** translated word to list of new words
  - **Make a string of the list** of new words
  - **Write** the string to the target file

lexicon.txt

```
1 a een
2 file bestand
3 is is
4 short kort
5 this dit
6 very zeer
```

english.txt

```
1 this is a
2 very short file|
```

Radboud Universiteit



# Example: Translate file and write to new file

```
11 lexicon = read_lexicon('lexicon.txt')  
12  
13 source_file = open('english.txt')  
14 target_file = open('dutch.txt', 'w')  
15 for line in source_file:  
16     words = line.strip().split()  
17     new_line = []  
18     for word in words:  
19         if word in lexicon:  
20             new_word = lexicon[word]  
21             new_line.append(new_word)  
22     target_file.write(' '.join(new_line)+'\n')  
23 target_file.close()  
24 source_file.close()
```

The open function creates a file handle named `target_file` that points to the file `'dutch.txt'` and opens it for writing. If this file does not yet exist, it will be created.

The write method write a string to the file handle `target_file`.

## Alternative:

```
for word in new_line:  
    target_file.write(word)  
target_file.write('\n')
```

# Files

- End of line character ('\n') at the end of a line
- You have to know the format of the file when you read and handle it
  - Many standardised formats (CSV, [JSON](#), XML, etc.)

```
lexicon.txt
1  a een
2  file bestand
3  is is
4  short kort
5  this dit
6  very zeer
```

```
english.txt
1  this is a
2  very short file
```

# Format operator (current version)

- Argument of write() has to be a string, two options:
  - type cast with str()
  - use format operator
- `"some text {} some more text".format(variable_of_any_type)`
- Examples:
  - `my_string = "{} degrees Celsius is {} degrees Fahrenheit".format(26, 78.800)`
  - `my_string = "{celsius} degrees Celsius is {fahrenheit} degrees Fahrenheit".format(celsius=26, fahrenheit=78.800)`
  - `my_string = "{} degrees Celsius is {:.1f} degrees Fahrenheit".format(26, 78.800)`
    - 78.800 => 78.8

```
>>> my_string
'26 degrees Celsius is 78.8 degrees Fahrenheit'
```

# Readings and practice for this week

- **Think Python Book:**
  - **Sections 9.1 + 14.1 + 14.2 (+ 14.3 + 14.4)**
  - **File handling:** [Link](#)
- **Practical video tutorials:**
  - Files: [Link](#)
  - Positional, keywords & default arguments: [Link](#)
  - (New) format operator: [Link](#)
- <https://coderbyte.com/> (View Challenges without the lock (unless you have a paid subscription))

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian.tejedorgarcia@ru.nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

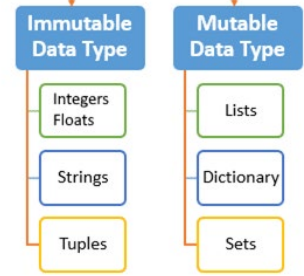
Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)



# Lecture 10:

## Tuples and more on Strings, Lists, Dictionaries & Sets

Programming for Beginners: Python



# Tuples *(practice at home - [link](#))*

- **Immutable** list: `(x, y)` or `x, y` instead of `[x, y]`
- A couple of (or more) variables together
- A function can in principle return only one value, but by using a tuple, you can “trick” Python into having a function return more than one value

```
person = "Xi Jinping"
```

```
(street, number, zip_code, city, state) = give_address(person)
```

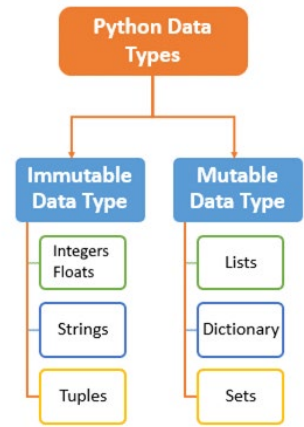
or

```
street, number, zip_code, city, state = give_address(person)
```

# String methods

- `count()`
- `find()` / `rfind()`
- `format()`
- `isalpha()` / `isdigit()` / etc.
- `isupper()` / `islower()`
- `lower()` / `upper()`
- `join()`
- `replace()`
- `strip()`
- `split()`
- etc., see [https://www.w3schools.com/python/python\\_ref\\_string.asp](https://www.w3schools.com/python/python_ref_string.asp)

All string methods return new values. They do not change the original string.



# String method **split()**

- Definition and Usage

- The split() method splits a string into a list.
- You can specify the separator, default separator is any whitespace.

- Syntax

- `string_variable.split(separator, maxsplit)`

- | Parameter        | Description  |
|------------------|--|
| <i>separator</i> | <b>Optional.</b> Specifies the separator to use when splitting the string. By <b>default</b> any whitespace is a separator |
| <i>maxsplit</i>  | <b>Optional.</b> Specifies how many splits to do. Default value is -1, which is "all occurrences"                          |

- ```
elements = "hello+world+!".split("+")
elements => ???
```

# String method split()

- Definition and Usage

- The split() method splits a string into a list.
- You can specify the separator, default separator is any whitespace.

- Syntax

- `string_variable.split(separator, maxsplit)`

- | Parameter        | Description                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------|
| <i>separator</i> | Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator |
| <i>maxsplit</i>  | Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences"                   |

- ```
elements = "hello+world+!".split("+")
elements => ["hello", "world", "!"]
```

# String method **join()**

- Definition and Usage

- The join() method takes all items in an iterable and joins them into one string.
- A string must be specified as the separator.

- Syntax

- `string_variable.join(iterable)`

- | Parameter       | Description  |
|-----------------|--|
| <i>iterable</i> | <b>Required.</b> Any iterable object where all the returned values are strings |

- ```
gym_accessories = ['towel', 'bottle of water', 'straps']  
glued = "&".join(gym_accessories)  
glued => ???
```

# String method join()

- Definition and Usage

- The join() method takes all items in an iterable and joins them into one string.
- A string must be specified as the separator.

- Syntax

- `string_variable.join(iterable)`

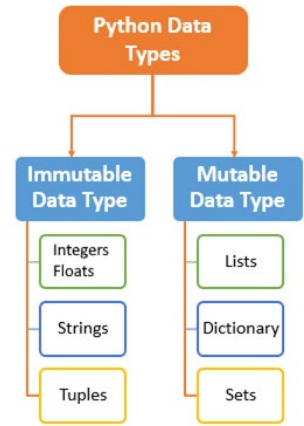
- | Parameter       | Description                                                             |
|-----------------|-------------------------------------------------------------------------|
| <i>iterable</i> | Required. Any iterable object where all the returned values are strings |

- ```
gym_accessories = ['towel', 'bottle of water', 'straps']  
glued = "&".join(gym_accessories)  
glued => 'towel&bottle of water&straps'
```

# List methods

- `append()`
- `clear()`
- `count()`
- `pop()`
- `remove()`
- `reverse()`
- `sort()`
- etc., see [https://www.w3schools.com/python/python\\_ref\\_list.asp](https://www.w3schools.com/python/python_ref_list.asp)

List methods can change the list, return values or both.





# List method **append()**

- Definition and Usage

- The `append()` method appends an element to the end of the list.

- Syntax

- `list_variable.append(element)`

- 

Parameter	Description
<i>element</i>	<b>Required.</b> An element of any type (string, number, object etc.)

- ```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1.append(list2)
list3 => ???
```

# List method append()

- Definition and Usage

- The append() method appends an element to the end of the list.

- Syntax

- `list_variable.append(element)`

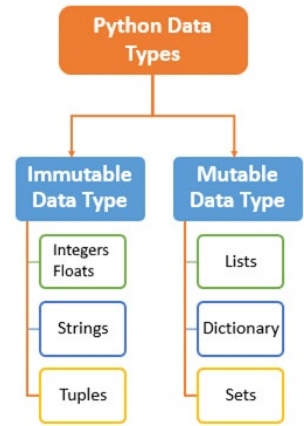
- 

| Parameter      | Description                                                    |
|----------------|----------------------------------------------------------------|
| <i>element</i> | Required. An element of any type (string, number, object etc.) |

- ```
list1 = [1,2,3]
list2 = [4,5,6]
list3 = list1.append(list2)
list3 => [1,2,3,[4,5,6]]
```

# Dictionary methods

- `clear()`
  - `copy()`
  - `keys()`
  - `pop()`
  - `values()`
  - etc., see [https://www.w3schools.com/python/python\\_ref\\_dictionary.asp](https://www.w3schools.com/python/python_ref_dictionary.asp)
- Dictionary methods can change the dictionary, return values or both.
- Remember: dictionaries are not ordered



# Dictionary method **pop()**

- Definition and Usage

- The pop() method removes the specified item from the dictionary.
- The value of the removed item is the return value of the pop() method.

- Syntax

- `dictionary_variable.pop(keyname, defaultvalue)`

- | Parameter           | Description   |
|---------------------|---|
| <i>keyname</i>      | <b>Required.</b> The keyname of the item you want to remove   |
| <i>defaultvalue</i> | Optional. A value to return if the specified key does not exist.<br>If this parameter is not specified, and no item with the specified key is found, an error is raised |

- ```
prices = {'bread': '€2', 'zoo-ticket': '€20', 'car': '€20,000', 'house': '€200,000'}  
removed_item = prices.pop('zoo-ticket')  
prices = ???          /          removed_item = ???
```

# Dictionary method pop()

- Definition and Usage

- The pop() method removes the specified item from the dictionary.
- The value of the removed item is the return value of the pop() method.

- Syntax

- `dictionary_variable.pop(keyname, defaultvalue)`

- | Parameter           | Description                                                                                                                                                               |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>keyname</i>      | Required. The keyname of the item you want to remove                                                                                                                      |
| <i>defaultvalue</i> | Optional. A value to return if the specified key do not exist.<br>If this parameter is not specified, and the no item with the specified key is found, an error is raised |

- ```
prices = {'bread': '€2', 'zoo-ticket': '€20', 'car': '€20,000', 'house': '€200,000'}  
removed_item = prices.pop('zoo-ticket')  
prices = {'bread': '€2', 'car': '€20,000', 'house': '€200,000'}
```

# Dictionary method pop()

- Definition and Usage

- The pop() method removes the specified item from the dictionary.
- The value of the removed item is the return value of the pop() method.

- Syntax

- `dictionary_variable.pop(keyname, defaultvalue)`

- | Parameter           | Description   |
|---------------------|---|
| <i>keyname</i>      | Required. The keyname of the item you want to remove  |
| <i>defaultvalue</i> | Optional. A value to return if the specified key do not exist.<br>If this parameter is not specified, and the no item with the specified key is found, an error is raised |

- ```
prices = {'bread': '€2', 'zoo-ticket': '€20', 'car': '€20,000', 'house': '€200,000'}  
removed_item = prices.pop('zoo-ticket')  
removed item = '€20'
```

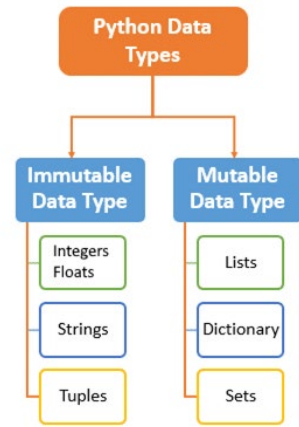
# Set methods [\(link\)](#)

- `lis = range(10)`
- `A = set(lis)`
- `print(A)`
- `print(10 in A)`
- `print(10 not in A)`
- `B = set(range(6, 20))`
- `print(B)`
- `print(A.issubset(B))` #subset
- `print(A.issuperset(B))` # superset
- `print(A | B)`
- `print(A & B)`
- `print(A - B)`
- `print(A ^ B)` # not common elems.

- `A.add(0)`
- `A.discard(2)`
- `A.clear()`

- # 1. Copy by **reference** (problems)
- `B = A`
- `print(A)`
- `print(B)`
- `A.add(666)`
- `print(A)`
- `print(B)`

- # 2. Copy by **value** (less efficient)
- `C = A.copy()`
- `print(A)`
- `print(C)`
- `A.add(777)`
- `print(A)`
- `print(C)`



# Example: Remove obsolete words from dictionary

- We have a dictionary in a text file
1. When the dictionary is too large, we have to **remove words**
  2. We have a list **with obsolete words**, that can be removed from the dictionary
  3. When we have removed words, we have to **write the new dictionary** to a new file



# Example: Remove obsolete words from dictionary

dictionary.txt

```
1  Lemma;Pronunciation;POS;Meaning(s)
2  bide;baid;verb;remain somewhere
3  harissa;hə'ris.ə;noun;a spicy sauce or pasta
4  pence;pens;noun;plural of penny
5  trump;tramp;noun;a playing card;horn
```

obsoletes.txt

```
1  trump
2  pence
```

# Example: Remove obsolete words from dictionary

- **In pseudocode:**
  - Read dictionary
  - If dictionary is too large
    - Check for obsolete words and remove these
    - Write updated dictionary to new file

# Example: Remove obsolete words from dictionary

```
32 max_dict_size = 2
33 old_dict, dict_size = read_dictionary('dictionary.txt')
34 if dict_size > max_dict_size:
35     obsoletes = read_obsoletes('obsoletes.txt')
36     new_dict = remove_obsoletes(old_dict, obsoletes, max_dict_size)
37     write_dictionary(new_dict, 'dictionary_new.txt')
```

# Example: Remove obsolete words from dictionary

```
32 max_dict_size = 2
33 old_dict, dict_size = read_dictionary('dictionary.txt')
34 if dict_size > max_dict_size:
35     obsoletes = read_obsoletes('obsoletes.txt')
36     new_dict = remove_obsoletes(old_dict, obsoletes, max_dict_size)
37     write_dictionary(new_dict, 'dictionary_new.txt')
```

Return value of function  
`read_dictionary` is a tuple.

# Example: Create function that **reads dictionary**

- **In pseudocode:**

- Let "dictionary" be an empty data structure that pairs the lemmas with pronunciation, part of speech tags and meanings
- Let "count" be a counter that keeps the number of dictionary entries
- Open the dictionary file
- For each line in the dictionary file:
  - Skip the header line
  - Remove the newline character
  - Split the line in a maximum of four fields: lemma, pronunciation, pos, meaning
  - Store the pronunciation, pos and meaning as value of key lemma in "dictionary"
  - Increase "count" with 1
- Return "dictionary" and "count"

## Example: Create function that reads dictionary

```
1 def read_dictionary(filename):
2     dictionary = {}
3     count = 0
4     dictionary_file = open(filename)
5     for line in dictionary_file:
6         (lemma,pron,pos,meaning) = line.strip().split('; ',3)
7         if not lemma == 'Lemma':
8             dictionary[lemma] = (pron,pos,meaning)
9             count += 1
10    dictionary_file.close()
11    return(dictionary,count)
```

dictionary.txt

```
1 Lemma;Pronunciation;POS;Meaning(s)
2 bide;baid;verb;remain somewhere
3 harissa;hə'ris.ə;noun;a spicy sauce or pasta
4 pence;pens;noun;plural of penny
5 trump;tramp;noun;a playing card;horn
```

# Example: Create function that reads dictionary

```
1 def read_dictionary(filename):
2     dictionary = {}
3     count = 0
4     dictionary_file = open(filename)
5     for line in dictionary_file:
6         (lemma,pron,pos,meaning) = line.strip().split('; ',3)
7         if not lemma == 'Lemma':
8             dictionary[lemma] = (pron,pos,meaning)
9             count += 1
10    dictionary_file.close()
11    return(dictionary,count)
```

Separation character of split is semi-colon.  
Maximum number of splits is 3 (yielding 4 fields).

(lemma,pron,pos,meaning) = line.strip().split('; ',3)

dictionary[lemma] = (pron,pos,meaning)

Value of dict `dictionary` is a tuple.

Return value is a tuple.

# Example: Create function that **reads obsoletes file**

- **In pseudocode:**

- Let "obsoletes" be an empty data structure that stores the obsolete words
- Open the obsoletes file
- For each line in the obsoletes file:
  - Remove the newline character
  - Store the word in "obsoletes"
  - Return "obsoletes"

obsoletes.txt

```
1 trump
2 pence
```



## Example: Create function that reads obsoletes file

```
13 def read_obsoletes(filename):  
14     obsoletes = []  
15     obsoletes_file = open(filename)  
16     for line in obsoletes_file:  
17         obsoletes.append(line.strip())  
18     return obsoletes
```

obsoletes.txt

```
1 trump  
2 pence
```

# Example: Create function that **removes obsoletes**

- **In pseudocode:**
  - As long as the dictionary is too big
    - Take an obsolete word that has not yet been deleted
    - Remove it from the dictionary
    - Return the dictionary

## Example: Create function that removes obsoletes

```
20 def remove_obsoletes(dictionary, obsolete_words, max_size):  
21     while len(dictionary) > max_size:  
22         dictionary.pop(obsolete_words.pop(), "nonexistent")  
23     return dictionary
```

# Example: Create function that removes obsoletes

```
20 def remove_obsoletes(dictionary, obsolete_words, max_size):  
21     while len(dictionary) > max_size:  
22         dictionary.pop(obsolete_words.pop(), "nonexistent")  
23     return dictionary
```

First we pop a word from the list `obsolete_words`, then we pop that word from the dictionary `dictionary`, Note the second argument in the `pop` function of the dictionary that prevents an error message when the word is not in the dictionary

# Example: Create function **write the new dictionary**

- **In pseudocode:**

- Open a dictionary file for writing
- Write the header line
- For each entry in the dictionary
  - Write the lemma, pronunciation, pos and meaning

## Example: Create function write the new dictionary

```
25 def write_dictionary(dictionary,filename):
26     dictionary_file = open(filename,mode='w')
27     dictionary_file.write('Lemma;Pronunciation;POS;Meaning(s)\n')
28     for lemma in dictionary:
29         dictionary_file.write(lemma+';'+';'.join(dictionary[lemma])+'\n')
30     dictionary_file.close()
```

dictionary.txt

```
1 Lemma;Pronunciation;POS;Meaning(s)
2 bide;baid;verb;remain somewhere
3 harissa;hə'ris.ə;noun;a spicy sauce or pasta
4 pence;pens;noun;plural of penny
5 trump;tramp;noun;a playing card;horn
```

# Practice at home

- Try to code the same example (slides 16-30) in **replit**
- Using two similar files as in the example:
  - `dictionary.txt` and `obsoletes.txt`
- **If you have any question, send us an email**

# Readings/Tutorials for this week

- **Think Python Book:**
  - **Chapter 12**
- **Youtube tutorials**

Lists, Tuples, and Sets: [Link](#)

Intermediate Python Programming Course: [Link](#)
- **www.w3schools.com**
  - [https://www.w3schools.com/python/python\\_ref\\_string.asp](https://www.w3schools.com/python/python_ref_string.asp)
  - [https://www.w3schools.com/python/python\\_ref\\_list.asp](https://www.w3schools.com/python/python_ref_list.asp)
  - [https://www.w3schools.com/python/python\\_ref\\_dictionary.asp](https://www.w3schools.com/python/python_ref_dictionary.asp)
  - [https://www.w3schools.com/python/python\\_ref\\_set.asp](https://www.w3schools.com/python/python_ref_set.asp)
- **Resubmit the exercises you found difficulties (previous assignments)**



# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian.tejedorgarcia@ru.nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Lecture 11:

# Modules

Programming for Beginners: Python

Radboud Universiteit



# Modules

- Python code that you can import in your script
  - A couple of modules together is called a *software package*

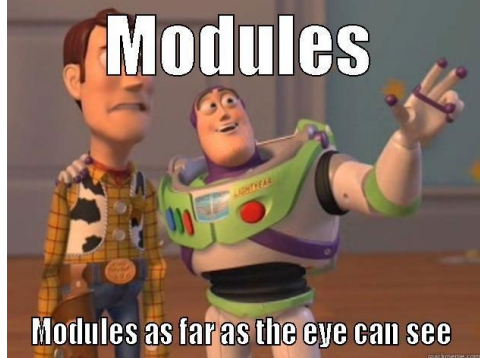
## import

- Python reads the script and interprets it, but knows where it comes from (name)
- The *name* of the module can be changed or skipped
- Leave out `.py` when importing

```
import <module>
import <module> as <md>
from <module> import <function(s)>
array
from <module> import <variable(s)>
from <module> import *
numpy import *
```

```
# import numpy
# import numpy as np
# from numpy import
```

Radboud Universiteit



# Import module

sounds.py

```
1  animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}
2
3  def animal_talk(animal):
4      | print('the {} says {}'.format(animal,animal_sounds[animal]))
```

# Import module

sounds.py

```
1  animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}  
2  
3  def animal_talk(animal):  
4      | print('the {} says {}'.format(animal,animal_sounds[animal]))
```

main.py

```
1  import sounds  
2  
3  for animal in sounds.animal_sounds:  
4      | sounds.animal_talk(animal)
```

# Import module

sounds.py

```
1 animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}
2
3 def animal_talk(animal):
4     print('the {} says {}'.format(animal,animal_sounds[animal]))
```

main.py

```
1 import sounds
2
3 for animal in sounds.animal_sounds:
4     sounds.animal_talk(animal)
```

we have to put `sounds` before  
the object/variable or function

# Import module

sounds.py

```
1 animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}
2
3 def animal_talk(animal):
4     print('the {} says {}'.format(animal,animal_sounds[animal]))
```

main.py

```
1 import sounds
2
3 for animal in sounds.animal_sounds:
4     sounds.animal_talk(animal)
```

```
the cat says meow
the duck says quack
the cow says moo
```



# Import module (as)

sounds.py

```
1 animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}
2
3 def animal_talk(animal):
4     print('the {} says {}'.format(animal,animal_sounds[animal]))
```

main.py

```
1 import sounds as snds
2
3 for animal in snds.animal_sounds:
4     snds.animal_talk(animal)
```

sounds is now called snds

snds must be in front of object or function.

```
the cat says meow
the duck says quack
the cow says moo
```





# Import module (**from**)

sounds.py

```
1 animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}
2
3 def animal_talk(animal):
4     print('the {} says {}'.format(animal,animal_sounds[animal]))
```

main.py

```
1 from sounds import animal_sounds, animal_talk
2
3 for animal in animal_sounds:
4     animal_talk(animal)
```

import from `sounds`

nothing in front of object or function

```
the cat says meow
the duck says quack
the cow says moo
>
```

# Import module (from,as)

sounds.py

```
1 animal_sounds = {'cat':'meow','duck':'quack','cow':'moo'}
2
3 def animal_talk(animal):
4     print('the {} says {}'.format(animal,animal_sounds[animal]))
```

main.py

```
1 from sounds import animal_sounds as a_s, animal_talk as a_t
2
3 for animal in a_s:
4     a_t(animal)
```

```
the cat says meow
the duck says quack
the cow says moo
>
```

# Standard utility modules

- Python has a couple of standard modules
  - very common/useful
  - if you need a little bit more than the built-in functions (which is pretty soon)
  - come with every Python installation (other modules must be installed separately)
- Some useful standard modules:
  - `random`
  - `datetime`
  - `collections`
  - `os.path`
  - `math`
  - `csv`
  - `re`

```
1 import random
2
3 # float between 0 and 1
4 print(random.random())           # 0.9288030517961269
5
6 # integer between 10 and 20
7 print(random.randint(10,20))     # 13
8
9 # integer between 0 and 100, with step 10
10 print(random.randrange(0,100,10)) # 70
11
12 animals = ['cat','duck','cow','deer','lion','penguin']
13
14 # pick one from a sequence (list)
15 print(random.choice(animals))    # cat
16
17 # pick one from a sequence (string)
18 print(random.choice('abracadabra')) # a
19
20 # pick four from a sequence
21 print(random.sample(animals,4))   # ['deer', 'penguin', 'duck', 'cat']
22
23 # shuffle a sequence
24 random.shuffle(animals)
25 print(animals)                   # ['cat', 'lion', 'cow', 'penguin', 'duck', 'deer']
```

```

1  from datetime import datetime, date, time
2
3  # now
4  print(datetime.now())                # 2020-11-18 09:53.709719
5
6  # define date
7  motmdate = date(1969,7,16)
8  # define time
9  motmtime = time(14,32)
10 # combine date and time
11 motmdatetime = datetime.combine(motmdate,motmtime)
12
13 # how long ago was the first man on the moon exactly
14 print(datetime.now()-motmdatetime)    # 18752 days, 18:24:22.005832
15
16 # fancy printing
17 print(motmdatetime.strftime("%A, %d. %B %Y at %I:%M%p"))
18 | | | | | | | | | | | | | | | | # Wednesday, 16. July 1969 at 02:32PM
19
20 # datestring in dd/mm/yyyy HH:MM:SS
21 datetime_string = "17/07/1969 14:32:00"
22 # solve datetime encoding
23 datetime_solved = datetime.strptime(datetime_string, "%d/%m/%Y %H:%M:%S")
24 print(datetime_solved)                # 1969-07-17 14:32:00

```

```
1 from collections import Counter
2
3 # define string
4 text1 = 'Remember when you were young, '
5 # define list
6 text2 = ['y','o','u',' ','s','h','o','n','e',' ','l','i','k','e',' ','t','h','e',
7         ' ','s','u','n','.']
8
9 # create counters
10 t1 = Counter(text1)
11 t2 = Counter(text2)
12 print(t1)
13 # Counter({'e': 6, ' ': 5, 'm': 2, 'r': 2, 'w': 2, 'n': 2, 'y': 2, 'o': 2, 'u':
14 2, 'R': 1, 'b': 1, 'h': 1, 'g': 1, ',': 1})
15
16 print(t2)
17 # Counter({' ': 4, 'e': 3, 'o': 2, 'u': 2, 's': 2, 'h': 2, 'n': 2, 'y': 1, 'l':
18 1, 'i': 1, 'k': 1, 't': 1, '.': 1})
19
20 # frequently top 5
21 print(t1.most_common(5))
22 # [('e', 6), (' ', 5), ('m', 2), ('r', 2), ('w', 2)]
23
24 # subtract occurrences of two counters
25 t1.subtract(t2)
26 print(t1.most_common(5))
27 # [('e', 3), ('m', 2), ('r', 2), ('w', 2), ('R', 1)]
```

```
27 from collections import deque
28
29 # create deque from string
30 d1 = deque('abcdefghijklmnopqrstuvwxyz')
31 print(d1)
32 # deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
    'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'])
33
34 # rotate with 3
35 d1.rotate(3)
36 print(d1)
37 # deque(['x', 'y', 'z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
    'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w'])
38
39 d2=deque([4,5,6,7,8,9,10])
40 # append to the front
41 d2.appendleft(3)
42 d2.appendleft(2)
43 d2.appendleft(1)
44 print(d2)
45 # deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
1  import os.path
2
3  pathstring = '/home/users/eric/python/course/lecture/9/examples.py'
4
5  # file name
6  print(os.path.basename(pathstring))
7  # examples.py
8
9  # directory name
10 print(os.path.dirname(pathstring))
11 # /home/users/eric/python/course/lecture/9
12
13 # check if file
14 print(os.path.isfile(pathstring))      # False
15
16 # check if directory
17 print(os.path.isdir(pathstring))      # False
18
19 # check if exists at all
20 print(os.path.exists(pathstring))     # False
21 #
22
23 # split extension
24 dirfile, extension = os.path.splitext(pathstring)
25 print(dirfile, extension)
```



```
1  import math
2
3  # pi
4  print(math.pi)
5  # 3.141592653589793
6
7  # multiply all elements in iterable
8  print(math.prod([1,2,3,4,5]))
9  # 120
10
11 # cosine
12 print(math.cos(5))
13 # 0.28366218546322625
14
15 # round down
16 print(math.floor(10*math.cos(5)))
17 # 2
18
19 # round up
20 print(math.ceil(10*math.cos(5)))
```

# Readings and practice for this week

- **Think Python 2 Book:**
  - Section 14.9
- **Practical tutorials:**
  - **Modules:** [Link](#)
  - Practice with more challenges: [Link](#)
- **docs.python.org**
  - <https://docs.python.org/3/library/> (browse, don't study)
- **Resubmit the exercises you found difficulties (previous assignments)**

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian . tejedorgarcia @ ru . nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Lecture 12:

# Regular Expressions

Programming for Beginners: Python

Radboud Universiteit



# Introduction

```
>>> s = 'foo123bar'
>>> '123' in s
True
```

```
>>> s = 'foo123bar'
>>> s.find('123')
3
>>> s.index('123')
3
```

How to determine whether a string contains any three consecutive decimal digit characters as in:

*'foo123bar', 'foo456bar',  
'234baz', 'qux678'*

?

# Regular Expressions (*regex*)

- A regular expression is a sequence of characters that can be used to define a search pattern ([Wikipedia](https://en.wikipedia.org/wiki/Regular_expression)).
- Regular expressions are used to find patterns in a text.
- Simplest regex is **literal text**:



`Hello`

matches

`Hello, World!`

`#&56@%`

matches

`12#$34#&56@%78) (90!?`

`hell`

does **not** match

`Hello, World!`

`1234567890`

does **not** match

`12#$34#&56@%78) (90!?`

# Options

- Set (any of)
- Range
- Predefined set
- Number
- Wildcard
- Complementing (anything but)
- Alternation (or)
- Begin and End
- Group
- Escape

[ ]  
-  
\  
\* + ?  
.  
^  
|  
^ and \$  
( )  
\  
u



# Set []

- match if any of the characters between the brackets match

`[abcWxyz]` matches `Hello, World!`

`[123456789]` matches `12#$34#&56@%78)(90!?`

`[ABCwXYZ]` does **not** match `Hello, World!`

`[abcxyz]` does **not** match `12#$34#&56@%78)(90!?`



# Range -

- makes a range of characters

**a-z** equals **abcdefghijklmnopqrstuvwxyz**

**A-Z** equals **ABCDEFGHIJKLMNOPQRSTUVWXYZ**

**0-9** equals **0123456789**

**[A-Z]** matches **Hello, World!**

**[0-9]** matches **12#\$34#&56@%78) (90!?**

# Predefined set \

- shorthand for a group of similar characters together
  - `\d` matches any digit, equals `[0-9]`
  - `\D` matches anything but a digit
  - `\w` matches any alphanumeric character, equals `[a-zA-Z0-9_]`
  - `\W` matches anything but a alphanumeric character
  - `\s` matches any whitespace character, equals `[\t\n\r\f\v]`
  - `\S` matches anything but whitespace character

`[\D], [\w], [\W], [\s], [\S]` all match **Hello, World!**

`[\d]` does **not** match **Hello, World!**

`[\d], [\D], [\w], [\W], [\S]` all match **12#\$34#&56@%78) (90!?**

`[\s]` does **not** match **12#\$34#&56@%78) 9 (0!?**

# Number \* + ? { }

- define how many elements should match ([link to a useful comparison](#))

\* matches zero or more elements

+ matches one or more elements

? matches zero or one element

{**m**} matches exactly **m** consecutive elements

{**m**,} matches **m** consecutive elements or more

{**m**,**n**} matches from **m** to **n** consecutive elements

**x**\* matches **Hello, World!**

**x**+ does **not** match **Hello, World!**

**#**? matches **12#\$34#&56@%78) (90!?**

**#**{**2**} does **not** match **12#\$34#&56@%78) (90!?**

**#**{**2**,} matches **12#####\$34#&56@%78) (90!~**

**1**{**1**,**5**} matches **He**l**lo, Wor**l**d!**

Radboud Universiteit



# Wildcard .

- . matches **any** character

.**\*** matches **Hello, World!**

.**+** matches **Hello, World!**

.**?** matches **12#\$34#&56@%78) (90!?**

.**{25}** does **not** match **12#\$34#&56@%78) (90!?**

.**{1,5}** matches **Hello, World!**

# Complement ^

- used as first character between brackets matches anything except the element(s) that follow

`[^x]` matches **Hello, World!**

`[^de]` matches **Hello, World!**

`[^\d]` does **not** match **1234567890**

`[^\d]{3}` does **not** match **12#\$34#&56@%78) (90!?**

# Alternation |

- match if any of the elements match

`h|H` matches `Hello, World!`

`-|:|,` matches `Hello, World!`

`a|b|c` does **not** match `12#$34#&56@%78) (90!?`

`[^\d]{3}|[^\D]{3}` does **not** match `12#$34#&56@%78) (90!?`

# Begin ^ and end \$ of string

- match if expression matches at begin ^ / end of string \$

`^H` matches `Hello, World!`

`ld!$` matches `Hello, World!`

`^2#` does **not** match `12#$34#&56@%78) (90!?`

`^[\\d]{2}.*[^\\D]{3}$` does **not** match `12#$34#&56@%78) (90!?`

# Group ( )

- groups elements together
- can be used to get the exact match

`(e1)` matches **He**llo, World!

`(\d\d)+` matches **12**#\$**34**#&**56**@%**78**) (**90**!?

`(e1){2,3}` does **not** match **He**llo, World!

`(\d\d){5}` does **not** match **12**#\$**34**#&**56**@%**78**) (**90**!?



# Some examples

- Regular expressions should always match a consecutive set of characters/symbols
- But it can match on several places in a string

**A{3}** does not match **AbAbAb**

**Ab{3}** does not match **AbAbAb**

**(Ab){3}** matches **AbAbAb**

**(A.){3}** matches **AbAbAb**

**A{3}** matches **AAAb**lablabla**AAAb**lablabla**AAA**

# Escape \

- to match characters that are meaningful in regular expressions

`#$` does **not** match `12#$34#&56@%78)(90!?`

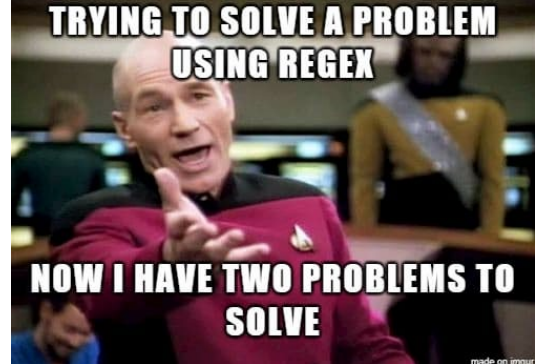
`#\$` matches `12#$34#&56@%78)(90!?`

`) (` does **not** match `12#$34#&56@%78)(90!?` (*gives an error even!*)

`\) \ (` matches `12#$34#&56@%78) (90!?`

# *re* module

- Regular expressions can be used with the module *re*
- <https://docs.python.org/3/library/re.html>
- functions:
  - `findall()`
  - `split()`
  - `sub()`
  - `search()`
  - ...



# re.findall(*re*,*string*)

- returns a **list** with all matches

```
1  import re
2
3  text = "abracadabra"
4  pattern = ".a"
5  matches = re.findall(pattern, text)
6  print(matches)
```

# re.findall(*re*,*string*)

- returns a list with all matches

```
1  import re
2
3  text = "abracadabra"
4  pattern = ".a"
5  matches = re.findall(pattern, text)
6  print(matches)
```

```
['ra', 'ca', 'da', 'ra']
```

## re.split(*re*,*string*)

- returns a **list** with parts after splitting on *re*

```
1  import re
2
3  text = "abracadabra"
4  pattern = "\wr"
5  parts = re.split(pattern,text)
6  print(parts)
```

# re.split(*re*,*string*)

- returns a list with parts after splitting on *re*

```
1  import re
2
3  text = "abracadabra"
4  pattern = "\br"
5  parts = re.split(pattern,text)
6  print(parts)
```

```
['a', 'acada', 'a']
```

```
a br acada br a
```

## re.sub(*re*,*sub*,*string*)

- returns a string with the *re* substituted by *sub* in *string*
- Similar to JavaScript `string.replace(pattern_expression, sub)`

```
1  import re
2
3  text = "abracadabra"
4  pattern = "[abcd]"
5  substitute = "r"
6  new_text = re.sub(pattern,substitute,text)
7  print(new_text)
```



## re.sub(*re*,*sub*,*string*)

- returns a string with the *re* substituted by *sub* in *string*
- Similar to JavaScript `string.replace(pattern_expression, sub)`

```
1  import re
2
3  text = "abracadabra"
4  pattern = "[abcd]"
5  substitute = "r"
6  new_text = re.sub(pattern,substitute,text)
7  print(new_text)
```

rrrrrrrrrrrrrrrr

## re.search(*re*,*string*)

- returns a Match object with the first match of *re* in *string*

```
1  import re
2
3  text = "abracadabra"
4  pattern = "(b.a)"
5  match = re.search(pattern, text)
6  print(match)
```

```
<re.Match object; span=(1, 4), match='bra'>
```

# Match object

- methods
  - `.span()`
    - returns a tuple with first and last index of match
  - `.group()`
    - returns exact match(es)
  - `.groups()`
    - returns list of exact match(es)

# Match object

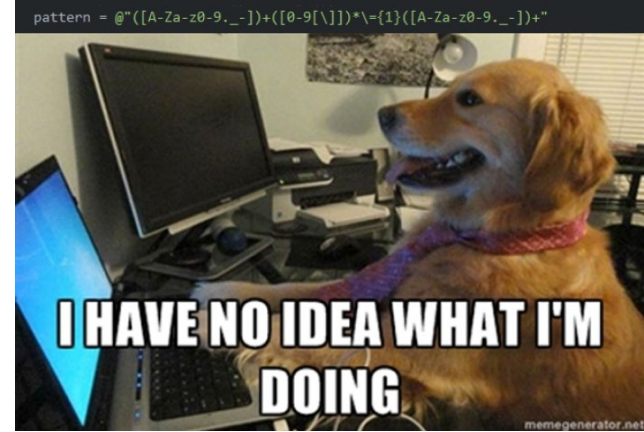
```
1 import re
2
3 text = "abracadabra"
4 pattern = "(a\\w).+([abc]a).+(a$)"
5 match = re.search(pattern, text)
6 print(match.span())
7 print(match.group(0))
8 print(match.group(1))
9 print(match.group(2))
10 print(match.group(3))
11 print(match.groups())
```

*A for/while loop is always a better solution :)*

```
>>> print(match.span())
(0, 11)
>>> print(match.group(0))
abracadabra
>>> print(match.group(1))
ab
>>> print(match.group(2))
ca
>>> print(match.group(3))
a
>>> print(match.groups())
('ab', 'ca', 'a')
```

# Useful extra information

- **Test you regexs here:**  
<https://regex101.com/>
- **Useful cheat sheet (PDF):** <https://www.dataquest.io/wp-content/uploads/2019/03/python-regular-expressions-cheat-sheet.pdf>
- **Most patterns/regex already exist (search on Google first):**  
<https://emailregex.com/>  
<https://stackoverflow.com/questions/17898523/regular-expression-for-dutch-zip-postal-code>  
<https://www.geeksforgeeks.org/python-regex-lookahead/>



# Readings this week



- **Practical tutorials (practice makes perfect)**

- <https://realpython.com/regex-python/>
- <https://www.youtube.com/watch?v=UQQsYXa1EHs>
- <https://www.youtube.com/watch?v=tnRXBBF8nO0>
- <https://www.youtube.com/watch?v=zN8rwVXwRUE>
- <https://www.google.com/search?q=regex+memes> :)

- **re module documentation:**

- <https://docs.python.org/3/howto/regex.html>
- [https://www.w3schools.com/python/python\\_regex.asp](https://www.w3schools.com/python/python_regex.asp)
- <https://docs.python.org/3/library/re.html>

- **Resubmit the exercises you found difficulties (previous assignments)**

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a virtual meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian.tejedorgarcia@ru.nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)

# Lecture 13:

# Debugging

Programming for Beginners: Python

Radboud Universiteit





# Errors

- Syntax error
- Runtime error
- Semantic and Logical error

# Syntax Error



- Often an error in one character (like a *typo*)
- Detected by Python before trying to run the script -> you can't run it
- Are always fatal (cannot be “handled”)
- Always gives an error message
  - error on the line mentioned or the line before
- Often easy to repair
- Examples:
  - indentation error
  - mismatched parentheses
  - colon missing after loop/conditional/function definition

# Runtime Error

- Detected by Python while running the script
- Not always fatal (can often be “handled”)
- Most of the times gives an error message
  - error on the line mentioned
  - possibly with long traceback
- Examples:
  - index error (e.g. trying to access non-existing element of a list)
  - type error (e.g. trying to add an `integer` and a `string`)
  - never ending loop (no error, but infinite loop!)

# Semantic and Logical Error

- Not detected by Python
- Not fatal (in the sense that the execution is halted)
- No error message
- But the **wrong output**
- Often difficult to repair -> print()
- Examples:
  - Summing instead of multiplying
  - Using the wrong variable
  - Loop too long

# Programming & Debugging

- Do not write your whole script and then run it, but write small parts and run to see whether it does what you expect.
- Understand and resolve error messages
- *print()*
- Google

# Example

- We have a text and we have to give for each of the letters a, e, i, o, u the percentage of words this letter appears in.

# Break up

- text = ...
- make lowercase
- for every word in the text (split)
  - for every letter in a, e, i, o, u
    - check if letter is in word
    - if so, increase counter letter-in-word
    - if not, increase counter letter-not-word
- for every letter
  - compute and print percentage

Ongeveer 2.440.000 resultaten (0,49 seconden)

✓ [stackoverflow.com](#) > questions > ho... ▼ [Vertaal deze pagina](#)

## How do I lowercase a string in Python? - Stack Overflow

24 jul. 2015 — Use `.lower()` - For example: `s = "Kilometer"` `print(s.lower())`. The official 2.x documentation is here: `str.lower()` The official 3.x documentation is ...

5 antwoorden

**PYTHON** - Uppercase and **Lowercase** - Stack ... 1 antwoord 14 mei 2017

**Python Lowercase** and Uppercase String - Stack ... 2 antwoorden 27 mei 2020

**python**: how to make list with all **lowercase** ... 2 antwoorden 24 okt. 2017

Does **python** see an empty string as uppercase or ... 1 antwoord 29 apr. 2020

[Meer resultaten van stackoverflow.com](#)

✓ [www.geeksforgeeks.org](#) > isupper-is... ▼ [Vertaal deze pagina](#)

## isupper(), islower(), lower(), upper() in Python and their ...

24 nov. 2020 — In **Python**, `islower()` is a built-in method used for string handling. The `islower()` methods returns "True" if all characters in the string are **lowercase**, ...

✓ [www.geeksforgeeks.org](#) > python-st... ▼ [Vertaal deze pagina](#)

## Python String lower() Method - GeeksforGeeks

3 dec. 2020 — `lower()` method converts all uppercase characters in a string into **lower** characters and returns it. Syntax : `string.lower()`. Parameters : The ...



```
1 text = "Do you come from a land down under? Where women glow and men plunder. Can't  
2 you hear, can't you hear the thunder? You better run, you better take cover."  
3 word_with_vowel = {}  
4 word_without_vowel = {}  
5  
6 for word in text.lower().split():  
7     for letter in 'aeiou':  
8         if letter in word  
9             word_with_vowel[letter] += 1  
10        else:  
11            word_without_vowel[letter] += 1  
12  
13 for letter in 'aeiou':  
14     print("{}: {}".format (letter,word_without_vowel[letter]/word_with_vowel[letter]))
```

```

1  text = "Do you come from a land down under? Where women glow and men plunder. Can't
2
3  word_with_vowel = {}
4  word_without_vowel = {}
5
6  for word in text.lower().split():
7      for letter in 'aeiou':
8          if letter in word
9              word_with_vowel[letter] += 1
10             else:
11                 word_without_vowel[letter] += 1
12
13  for letter in 'aeiou':
14      print("{}: {}".format (letter,word_without_vowel[letter]/word_with_vowel[letter]))

```

```

Traceback (most recent call last):
  File "main.py", line 1, in <module>
    import aeiou
  File "/home/runner/LPiP-Lecture-13/aeiou.py", line 8
    if letter in word
                    ^
SyntaxError: invalid syntax

```

```
> █
```

```

1  text = "Do you come from a land down under? Where women glow and men plunder. Can't
2
3  word_with_vowel = {}
4  word_without_vowel = {}
5
6  for word in text.lower().split():
7      for letter in 'aeiou':
8          if letter in word:
9              word_with_vowel[letter] += 1
10         else:
11             word_without_vowel[letter] += 1
12
13  for letter in 'aeiou':
14      print("{}: {}".format (letter,word_without_vowel[letter]/word_with_vowel[letter]))
15

```

```

Traceback (most recent call last):
  File "main.py", line 1, in <module>
    import aeiou
  File "/home/runner/LPiP-Lecture-13/aeiou.py", line 15
    ^
SyntaxError: unexpected EOF while parsing

```



```
1 text = "Do you come from a land down under? Where women glow and men plunder. Can't  
you hear, can't you hear the thunder? You better run, you better take cover."  
2  
3 word_with_vowel = {}  
4 word_without_vowel = {}  
5  
6 for word in text.lower().split():  
7     for letter in 'aeiou':  
8         if letter in word:  
9             word_with_vowel[letter] += 1  
10        else:  
11            word_without_vowel[letter] += 1  
12  
13 for letter in 'aeiou':  
14     print("{}: {}".format (letter,word_without_vowel[letter]/word_with_vowel[letter]))
```

```
Traceback (most recent call last):  
  File "main.py", line 1, in <module>  
    import aeiou  
  File "/home/runner/LPiP-Lecture-13/aeiou.py", line 11, in <module>  
    word_without_vowel[letter] += 1  
KeyError: 'a'
```

```
> █
```

Ongeveer 694.000 resultaten (0,32 seconden)

✓ realpython.com › python-keyerror ▼ [Vertaal deze pagina](#)

## Python KeyError Exceptions and How to Handle Them – Real ...

The **Python KeyError** is a type of `LookupError` exception and denotes that there was an issue retrieving the key you were looking for. When you see a `KeyError`, the ...

✓ realpython.com › lessons › how-ha... ▼ [Vertaal deze pagina](#)

## How to Handle a Python KeyError – Real Python

The ultimate goal is to stop unexpected **KeyError** exceptions from being raised. The usual solution is to use `.get()`: `# ages.py ages = ...`

✓ wiki.python.org › moin › KeyError ▼ [Vertaal deze pagina](#)

## KeyError - Python Wiki

20 nov. 2012 — User. Login. **Python** raises a **KeyError** whenever a `dict()` object is requested ( using the format `a = adict[key]`) and the key is not in the dictionary. If ...

### Mensen vragen ook

What is KeyError in Python?



How do I fix KeyError in Python?



How do I skip a KeyError in Python?



## What is KeyError in Python?



A **Python KeyError** exception is what is raised when you try to access a key that isn't in a dictionary ( dict ). **Python's** official documentation says that the **KeyError** is raised when a mapping key is accessed and isn't found in the mapping. ... The most common mapping in **Python** is the dictionary.



```

1  text = "Do you come from a land down under? Where women glow and men plunder. Can't
   you hear, can't you hear the thunder? You better run, you better take cover."
2
3  word_with_vowel = {}
4  word_without_vowel = {}
5
6  for word in text.lower().split():
7      for letter in 'aeiou':
8          if letter in word:
9              word_with_vowel[letter] += 1
10         else:
11             word_without_vowel[letter] += 1
12
13  for letter in 'aeiou':
14      print("{}: {}".format (letter,word_without_vowel[letter]/word_with_vowel[letter]))

```

```

Traceback (most recent call last):
  File "main.py", line 1, in <module>
    import aeiou
  File "/home/runner/LPiP-Lecture-13/aeiou.py", line 11, in <module>
    word_without_vowel[letter] += 1
KeyError: 'a'

```

```
1  text = "Do you come from a land down under? Where women glow and men plunder. Can't  
   you hear, can't you hear the thunder? You better run, you better take cover."  
2  
3  word_with_vowel = {}  
4  word_without_vowel = {}  
5  for letter in 'aeiou':  
6      word_with_vowel[letter] = 0  
7      word_without_vowel[letter] = 0  
8  
9  for word in text.lower().split():  
10     for letter in 'aeiou':  
11         if letter in word:  
12             word_with_vowel[letter] += 1  
13         else:  
14             word_without_vowel[letter] += 1  
15  
16  for letter in 'aeiou':  
17     print("{}: {}".format (letter, word_without_vowel[letter]/word_with_vowel[letter]))  
18
```





```
a: 2.625
e: 1.0714285714285714
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    import aeiou
  File "/home/runner/LPiP-Lecture-13/aeiou.py", line 17, in <module>
    print("{}: {}".format (letter,word_without_vowel[letter]/word_with
_vowel[letter]))
ZeroDivisionError: division by zero
> |
```

```
1 text = "Do you come from a land down under? Where women glow and men plunder. Can't  
you hear, can't you hear the thunder? You better run, you better take cover."  
2  
3 word_with_vowel = {}  
4 word_without_vowel = {}  
5 for letter in 'aeiou':  
6     word_with_vowel[letter] = 0  
7     word_without_vowel[letter] = 0  
8  
9 for word in text.lower().split():  
10     for letter in 'aeiou':  
11         if letter in word:  
12             word_with_vowel[letter] += 1  
13         else:  
14             word_without_vowel[letter] += 1  
15  
16 for letter in 'aeiou':  
17     print(letter)  
18     print(word_without_vowel[letter])  
19     print(word_with_vowel[letter])  
20     print("{}: {}".format (letter, word_without_vowel[letter]/word_with_vowel[letter]))
```

a  
21  
8  
a: 2.625  
e  
15  
14  
e: 1.0714285714285714  
i  
29  
0

Traceback (most recent call last):  
 File "main.py", line 1, in <module>  
 import aeiou  
 File "/home/runner/LPiP-Lecture-13/aeiou.py", line 20, in <module>  
 print("{}: {}".format (letter,word\_without\_vowel[letter]/word\_with\_vowel[letter]))  
ZeroDivisionError: division by zero



```

1 text = "Do you come from a land down under? Where women glow and men plunder. Can't
you hear, can't you hear the thunder? You better run, you better take cover."
2
3 word_with_vowel = {}
4 word_without_vowel = {}
5 for letter in 'aeiou':
6     word_with_vowel[letter] = 0
7     word_without_vowel[letter] = 0
8
9 for word in text.lower().split():
10     for letter in 'aeiou':
11         if letter in word:
12             word_with_vowel[letter] += 1
13         else:
14             word_without_vowel[letter] += 1
15
16 for letter in 'aeiou':
17     if word_with_vowel[letter] > 0:
18         print("{}: {}".format(letter, word_without_vowel[letter]/word_with_vowel[letter]))
19     else:
20         print("No words with vowel {}".format(letter))

```

```

a: 2.625
e: 1.0714285714285714
No words with vowel i
o: 1.4166666666666667
u: 2.2222222222222223

```

```

1 text = "Do you come from a land down under? Where women glow and men plunder. Can't
  you hear, can't you hear the thunder? You better run, you better take cover."
2
3 word_with_vowel = {}
4 word_without_vowel = {}
5 for letter in 'aeiou':
6     word_with_vowel[letter] = 0
7     word_without_vowel[letter] = 0
8
9 for word in text.lower().split():
10     for letter in 'aeiou':
11         if letter in word:
12             word_with_vowel[letter] += 1
13         else:
14             word_without_vowel[letter] += 1
15
16 for letter in 'aeiou':
17     print("{}: {}".format(letter, 100*word_with_vowel[letter]/(word_with_vowel[letter]
    +word_without_vowel[letter])))

```

```

a: 27.586206896551722 %
e: 48.275862068965516 %
i: 0.0 %
o: 41.37931034482759 %
u: 31.03448275862069 %

```



# What else should I learn after this course?

- Documenting Python code:
  - <https://realpython.com/documenting-python-code/>
- Ternary operators:
  - [https://book.pythontips.com/en/latest/ternary\\_operators.html](https://book.pythontips.com/en/latest/ternary_operators.html)
- Try/Except
  - [https://www.w3schools.com/python/python\\_try\\_except.asp](https://www.w3schools.com/python/python_try_except.asp)
- Classes/Objects
  - <https://www.youtube.com/watch?v=f0TrMH9s-VE>
- Testing in Python:
  - <https://realpython.com/python-testing/>
- Try to learn other languages: JavaScript, Java...
  - Search for Youtube tutorials



# Practice for this week

- **Think Python Book:**
  - Appendix A: Debugging
- **Youtube tutorials:**
  - <https://www.youtube.com/watch?v=ToPP5UGgJUM>
  - <https://realpython.com/invalid-syntax-python/>
  - Extra: [https://www.youtube.com/watch?v=7Vmik1M\\_ry0](https://www.youtube.com/watch?v=7Vmik1M_ry0)

# Office hours

- **What?**

Pass by my office in the **Erasmus Building-8.02** (or ask for a **virtual** meeting) to ask any questions you have about the course, or to work through previous assignment exercises or exercises from the book together

- **When?**

***Mondays*** 10:00 - 12:00 & 13:15 - 15:00

***Fridays*** 11:00 - 15:00

Other days are possible, but send me an email first: [cristian . tejedorgarcia @ ru . nl](mailto:cristian.tejedorgarcia@ru.nl)

- **Where?**

Erasmus Building, room **8.02** OR virtual meeting (*send an email before in both cases*)



# Extra Lecture:

# Classes vs Objects and other Terminology

Programming for beginners: Python

# Classes

- No explanation how to make them, but what they are
- Class is combination of variables and functions
- Often variables are accessible only via functions
  - e.g. not `xyz = class.variable` but `xyz = class.get_variable()`
- The “things” from the modules we discussed last week are mostly classes
- Class is a template, a definition of what it **can** contain or do
- Object is a class in use, with values for the variables

**Base Class**

**Dog**

Create instance



**Object**

**Bobby**

**Properties**

**Methods**

Color

Sit

Eye Color

Lay Down

Height

Shake

Length

Come

Weight

**Property Values**

**Methods**

Color: Yellow

Sit

Eye Color: Brown

Lay Down

Height: 17 in

Shake

Length: 35 in

Come

Weight: 24 pounds

# Class datetime.datetime

- Variables (properties / attributes)
  - year, month, day, hours, minutes, seconds, microseconds, tzinfo
- Functions (methods)
  - date(), time(), now(), strftime(), strptime(), etc.

```
import datetime
motmdatetime = datetime.datetime(year=1969, month=7, day=16,
    hour=14, minute=32)
motmdatetime.hour
> 14
motmdatetime.date()
> datetime.date(1969, 7, 16)
```

# Terminology

- **Variable Name** and **Value**
  - a value is **assigned** to a variable name. `weight = 100` is an **assignment**
- **Class** and **Object**
  - an object is **instantiated** from a class. `motmdate = date(1969,7,16)` is an **instantiation**
- **Integer** and **Float** are numbers, without or with decimal digits
- **Character** is a symbol (letter, number, punctuation and other)
  - string of length 1
- **String** is a **Sequence** of one or more characters (text)

# Terminology

- **List** is a **Sequence** of **Elements**
- **List** and **String** are **Iterable**, take characters/elements one at a time
  - List and String have **Indexes**, position in sequence that belongs to an element
  - List and String have **Slices**, sub part of sequence (e.g. list[3:6])
- **Dictionary** is an array with **Keys** mapped to **Values**

# Terminology

- **Expression** is code that produces a value
- **Statement** is instruction to perform a specific task
  - Expression
  - **Conditional Statement** (if)
  - **Loop** (for, while)
- **Function** is a sequence of statements belonging together
  - Can be **called** (invoked)
  - Can **return** a **return value**
  - **Arguments** can be **passed** through a function
  - **Parameter** is placeholder for argument (compare variable name and value)
  - **Method** is function belonging to an class/object

# How to break down a problem (into code)

## Hangman

- You have a list of five words  
`words = ['repetition', 'bicycle', 'announcement', 'subscription', 'blueprint']`
- If a player has played a word, it should not appear again, so a player can play five times
- In a game, let the player guess a letter with the input function
- A player can make six errors in guessing a letter. The seventh time (s)he has lost
- You do not have to draw the gallow and the hangman, but you can just print the number of errors after each guess
- You also have to print the word with the letters that are guessed so far in the right place, and in place of the letters not yet guessed an underscore (' '). E.g. ' e et t \_'



deal later with how to get a word and how to treat the list with five words  
first see how to play hangman with one word  
play the game in your head or on paper, step by step

let user guess letter

let user guess letter  
check if letter in word

let user guess letter  
check if letter in word  
if letter in word

let user guess letter

check if letter in word

if letter in word

    check where the letter is in the word

let user guess letter

check if letter in word

if letter in word

    check where the letter is in the word

    fill the letter in on the blanks

let user guess letter

check if letter in word

if letter in word

- check where the letter is in the word

- fill the letter in on the blanks

- check if all letters are guessed

let user guess letter

check if letter in word

if letter in word

- check where the letter is in the word

- fill the letter in on the blanks

- check if all letters are guessed

- if all letter are guessed



let user guess letter

check if letter in word

if letter in word

    check where the letter is in the word

    fill the letter in on the blanks

    check if all letters are guessed

    if all letter are guessed

        print congratulations. FINISH THIS PART

let user guess letter

check if letter in word

if letter in word

    check where the letter is in the word

    fill the letter in on the blanks

    check if all letters are guessed

    if all letter are guessed

        print congratulations. FINISH THIS PART

    otherwise continue

let user guess letter

check if letter in word

if letter in word

    check where the letter is in the word

    fill the letter in on the blanks

    check if all letters are guessed

    if all letter are guessed

        print congratulations. FINISH THIS PART

    otherwise continue

if letter not in word

```
let user guess letter
check if letter in word
if letter in word
    check where the letter is in the word
    fill the letter in on the blanks
    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue
if letter not in word
    add 1 to number of errors
```

```
let user guess letter
check if letter in word
if letter in word
    check where the letter is in the word
    fill the letter in on the blanks
    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue
if letter not in word
    add 1 to number of errors
    check if number of errors greater than 6
```

```
let user guess letter
check if letter in word
if letter in word
    check where the letter is in the word
    fill the letter in on the blanks
    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue
if letter not in word
    add 1 to number of errors
    check if number of errors greater than 6
    if number of errors greater than 6
```

```
let user guess letter
check if letter in word
if letter in word
    check where the letter is in the word
    fill the letter in on the blanks
    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue
if letter not in word
    add 1 to number of errors
    check if number of errors greater than 6
    if number of errors greater than 6
        print condolences. FINISH THIS PART
```

```
let user guess letter
check if letter in word
if letter in word
    check where the letter is in the word
    fill the letter in on the blanks
    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue
if letter not in word
    add 1 to number of errors
    check if number of errors greater than 6
    if number of errors greater than 6
        print condolences. FINISH THIS PART
    otherwise continue
```



```
let user guess letter
check if letter in word
if letter in word
    check where the letter is in the word
    fill the letter in on the blanks
    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue
if letter not in word
    add 1 to number of errors
    check if number of errors greater than 6
    if number of errors greater than 6
        print condolences. FINISH THIS PART
    otherwise continue
print word with letters guessed so far
```

*let user guess letter*

*let user guess letter*

```
guess = input("Guess letter ")
```

```
guess = input("Guess letter ")  
if letter in word
```

```
guess = input("Guess letter ")  
if letter in word
```

```
if guess in word:
```

```
word = "repetition"
```

```
guess = input("Guess letter ")
```

```
if letter in word
```

```
if guess in word:
```

```
word = "repetition"
guess = input("Guess letter ")
if guess in word:
    check where the letter is in the word
```

```
word = "repetition"
guess = input("Guess letter ")
if guess in word:
    check where the letter is in the word

    for letter in word:
        if guess == letter:
```



```
word = "repetition"
guess = input("Guess letter ")
if guess in word:
    for letter in word:
        if guess == letter:
            fill the letter in on the blanks
```

```
word = "repetition"  
guess = input("Guess letter ")  
if guess in word:  
    for i in range(len(word)):  
        if guess == word[i]:  
            fill the letter in on the blanks
```

```
word = "repetition"
guess = input("Guess letter ")
if guess in word:
    for i in range(len(word)):
        if guess == word[i]:
            fill the letter in on the blanks

            guess_word[i] = guess
```

```
word = "repetition"
```

```
guess_word = "_" * len(word)
```

```
guess = input("Guess letter ")
```

```
if guess in word:
```

```
    for i in range(len(word)):
```

```
        if guess == word[i]:
```

```
            fill the letter in on the blanks
```

```
            guess_word[i] = guess
```

```
word = "repetition"
```

```
guess_word = ["_"] * len(word)
```

```
guess = input("Guess letter ")
```

```
if guess in word:
```

```
    for i in range(len(word)):
```

```
        if guess == word[i]:
```

```
            fill the letter in on the blanks
```

```
            guess_word[i] = guess
```

```
word = "repetition"
guess_word = ["_"] * len(word)
guess = input("Guess letter ")
if guess in word:
    for i in range(len(word)):
        if guess == word[i]:
            guess_word[i] = guess
```

*check if all letters are guessed*

*if all letter are guessed*

*print congratulations. FINISH THIS PART*

*otherwise continue*

```
word = "repetition"
guess_word = ["_"] * len(word)
guess = input("Guess letter ")
if guess in word:
    for i in range(len(word)):
        if guess == word[i]:
            guess_word[i] = guess

    check if all letters are guessed
    if all letter are guessed
        print congratulations. FINISH THIS PART
    otherwise continue

if not "_" in guess_word:
    print("Congratulations")
word_found = True
```

```
word = "repetition"
guess_word = ["_"] * len(word)
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
    if not "_" in guess_word:
        print("Congratulations")
        word_found = True
```



```
word = "repetition"
guess_word = ["_"] * len(word)
word_found = False
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
    if not "_" in guess_word:
        print("Congratulations")
        word_found = True
```

```
word = "repetition"
guess_word = ["_"] * len(word)
word_found = False
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            word_found = True
```

*if letter not in word  
add 1 to number of errors*

```
word = "repetition"
guess_word = ["_"] * len(word)
word_found = False
no_errors = 0
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
    if not "_" in guess_word:
        print("Congratulations")
        word_found = True
```

*if letter not in word  
add 1 to number of errors*

```
if not guess in word:
    no_errors += 1
```

```
word = "repetition"
guess_word = ["_"] * len(word)
word_found = False
no_errors = 0
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            word_found = True
```

*if letter not in word*  
*add 1 to number of errors*

```
else:
```

```
    no_errors += 1
```

```
word = "repetition"
guess_word = ["_"] * len(word)
word_found = False
no_errors = 0
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            word_found = True
    else:
        no_errors += 1

    check if number of errors greater than 6
    if number of errors greater than 6
        print condolences. FINISH THIS PART
    otherwise continue
```

```
word = "repetition"
guess_word = ["_"] * len(word)
word_found = False
no_errors = 0
while not word_found:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            word_found = True
    else:
        no_errors += 1
        if no_errors > 6:
            print("You hang")
            game_over = True
```

```
word = "repetition"
guess_word = ["_"] * len(word)
game_over = False
no_errors = 0
while not game_over:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            game_over = True
    else:
        no_errors += 1
        if no_errors > 6:
            print("You hang")
            game_over = True
```

```
word = "repetition"
guess_word = ["_"] * len(word)
game_over = False
no_errors = 0
while not game_over:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            game_over = True
    else:
        no_errors += 1
        if no_errors > 6:
            print("You hang")
            game_over = True
```

*print word with letters guessed so far*



```
word = "repetition"
guess_word = ["_"] * len(word)
game_over = False
no_errors = 0
while not game_over:
    guess = input("Guess letter ")
    if guess in word:
        for i in range(len(word)):
            if guess == word[i]:
                guess_word[i] = guess
        if not "_" in guess_word:
            print("Congratulations")
            game_over = True
    else:
        no_errors += 1
        if no_errors > 6:
            print("You hang")
            game_over = True
print("".join(guess_word))
```

# Readings this week

- **Think Python Book:**

Download pdf here: <https://greenteapress.com/wp/think-python-2e/>

- Chapter 15

- **docs.python.org**

- <https://docs.python.org/3/glossary.html>

- **www.pythonforbeginners.com**

- <https://www.pythonforbeginners.com/python-glossary>