Learn how to programme (in Python)

Marion Weinzierl

Advanced Research Computing

@arc_du

March 26, 2020

2020-03-25

Learn how to programme (in Python)

Marion Weinzierl

Advanced Research Computing

@arc_du

March 26, 2020

Welcome to our programming course. This course will teach you the basics of computer programming. We use Python as programming language in this course, but we hope that you will be able to transfer your knowledge to other languages easily, once you got the hang of "thinking like a programmer". The basic principles are similar in most popular programming languages.

Before we start, just some words about how this course will be organised. [Explain how things are going to work today.]

Outline

There are many different ways of how to give a basic introduction into programming – and I will give you some alternative material on the next slide – but this is the order we will do it in: After giving a short intro into how to start, we will learn about data types, variables and basic instructions.

One major task that you probably want to use computer programs for is data processing. Therefore, we'll look at how to get data in and out of your program next. In order to efficiently handle the data, we will than learn the fundamental control statements and how to best reuse and structure our code.

Materials used and recommended

- Python Wiki - Python for Non-Programmers
- How to think like a Computer Scientist
- A Whirlwind Tour of Python
- Software Carpentry - Programming with Python

Here is a list of sources that I have used when I prepared this course. I have already sent you this list of materials in one of my emails. It is also great for self-study, and you can use it after this course to continue on your own. The first one is the Python Wiki, which is, as a website, a good resource of Python knowledge. The link I give you here is to the site "Python for Non-Programmers", that has an extensive list of materials to learn Python if you have not programmed in any language before. The next one, "How to think like a Computer Scientist", is my personal favorite. I used some of the structure and ideas of the first few chapters for this course. It really does not assume previous knowledge or any kind of technical background.

In contrast to that, "A Whirlwind Tour of Python" has a faster pace, and seems to be targeted at a more technically-minded audience. Finally, you might or might not have heard of the Software Carpentry. This is an organisation aimed at providing basic software training for everyone. They have a "train the trainers" approach, which means that they provide material and training courses for software carpentry trainers. Thankfully, you are also allowed to access and use their material for free. This training course here gives a Python introduction at hands of a practical data processing task.

Introduction

Course Objectives

By the end of this course you should know

- how a basic computer program is written and executed,
- what basic data types and control statements are,
- how to get and process user input and data,
- how to structure your code using functions,
- what can lead to your program not working, and what to do about it,
- where to find further resources to practice your Python programming.

Course Objectives

By the end of this course you should know

- how a basic computer program is written and executed,
- what basic data types and control statements are,
- how to get and process user input and data,
- how to structure your code using functions,
- what can lead to your program not working, and what to do about it,
- where to find further resources to practice your Python programming.

Let's talk about our course objectives now. My main objective is to enable you to start programming with this course. Of course I cannot teach you all there is to learn about programming. But I hope to give you a starting point. We will talk about how to get a simple computer program running, the basics in terms of data types, variables, if-statements and loops, getting data in and out of your program, functions and variable scope, and a bit about comments, error types and debugging. By the end of the course I hope you will be able to continue from there. Programming is a lot about learning by doing. There are tons of online (and offline) resources, and also forums (online and offline) where you can ask for help.

Programming and Programming Languages

Why do we want to program?

[Collect reasons why participants want to learn programming]

Programming and Programming Languages

Programming means: make the computer do the work for you!

Programming means: make the computer do the work for you!

With a computer program we tell the computer to do some of the work we cannot or do not want to do by hand.

Programming and Programming Languages

Programming means: make the computer do the work for you!

- Do the maths
- Boring repetitions
- Too complicated/extensive tasks
- Big data sets
- ...

Programming means: make the computer do the work for you!

- Do the maths
- Boring repetitions
- Too complicated/extensive tasks
- Big data sets
- ...

These can be small things, like looping over repetitive tasks, or computing some mathematical expression. Or we might want to process data sets of some sort, do some kind of agent-based or numerical simulations, make different software components work together,... Being able to program opens up a whole lot of possibilities!

Programming and Programming Languages

Steps:

- Write your code in high-level programming language.
- Translate into low-level (machine/assembly) language.
- Execute program.

Steps:

- Write your code in high-level programming language.
- Translate into low-level (machine/assembly) language.
- Execute program.

Now, how do we get the computer to do these things for us? This is what we ar doing: We write the program in a language that we as human beings (with some training) can understand. As the computer won't be able to understand the code that way, we then compile it, which means that we translate it into computer language. Then we execute the program and, given we have made no mistake, the computer will get working for us.

Programming and ~~Programming~~ Scripting Languages

Steps:

- Write your code in high-level programming language.
- Interpret code and execute directly

Steps:

- Write your code in high-level programming language.
- Interpret code and execute directly

Things can be made a little bit easier by using an interpreted scripting language instead of a compiled programming language.

Programming and ~~Programming~~ Scripting Languages

Steps:

- Write your code in high-level programming language.
- Interpret code and execute directly

Beware: Oversimplification!

Programming and ~~Programming~~ Scripting Languages

Steps:

- Write your code in high-level programming language.
- Interpret code and execute directly

Beware: Oversimplification!

Please note that I am oversimplifying here a bit with regards to the terminology. The point I want to make is that there are languages, such as Python, which we will be using in this course, that translate and execute the code directly, without us having to do a separate compilation step.

How do you start?

We go the easy way (so you can build it up from there):

- Choose Python.
- Skip the installation bit: Jupyter Notebooks.
- HOWEVER: You have to type yourselves! Don't copy-paste (yet)![1]
- Have a play.

[1] If you try to copy-paste from these slides, you will sometimes get a syntax error because of the characters used in the PDF.

How do you start?

We go the easy way (so you can build it up from there):

- Choose Python.
- Skip the installation bit: Jupyter Notebooks.
- HOWEVER: You have to type yourselves! Don't copy-paste (yet)![1]
- Have a play.

[1] If you try to copy-paste from these slides, you will sometimes get a syntax error because of the characters used in the PDF.

Now let's get started! What do we do? As I said, we choose Python, we have two reasons for this: Firstly, it is an interpreted language, which means we do not need to compile it each time that we change the code. Secondly, it is currently very popular for a number of reasons, and people have asked us for a Python course.

How do you start?

We go the easy way (so you can build it up from there):

- Choose Python.
- Skip the installation bit: Jupyter Notebooks.
- HOWEVER: You have to type yourselves! Don't copy-paste (yet)![1]
- Have a play.

[1] If you try to copy-paste from these slides, you will sometimes get a syntax error because of the characters used in the PDF.

We make things even easier for ourselves by using the Jupyter Notebooks infrastructure. However, we are not using real Jupyter Notebooks (I will come to that in a moment). The "Whirlwind Tour to Python" book offers Jupyter Notebooks, if you are interested in trying them after this course. The reason I am using this setup is that I want to take away the setup and installation hurdle from learning to program for you. But still want you to do the typing yourself, as I that is one crucial bit in programming: Getting it into your fingers, and thinking about every character you put into your program, as every character will make a difference. You will find that out very soon for yourselves!

How do you start?

We go the easy way (so you can build it up from there):

- Choose Python.
- Skip the installation bit: Jupyter Notebooks.
- HOWEVER: You have to type yourselves! Don't copy-paste (yet)![1]
- Have a play.

[1] If you try to copy-paste from these slides, you will sometimes get a syntax error because of the characters used in the PDF.

Therefore, I will encourage you repeatedly throughout this lesson to have a play with what you have just learned. Don't be afraid to try out things, don't be afraid to make mistakes. In fact, sometimes it is useful to try and break things to find out how they work, what is possible and what isn't. My collagues and myself are here to help you if you get stuck, or if you do not understand the reason for problems or errors that you might encounter.

How do you start?

We go the easy way (so you can build it up from there):

- Choose Python.
- Skip the installation bit: Jupyter Notebooks.
- HOWEVER: You have to type yourselves! Don't copy-paste (yet)![1]
- Have a play.

[1] If you try to copy-paste from these slides, you will sometimes get a syntax error because of the characters used in the PDF.

Please try and reduce the size of this window now and get a window with a web browser next to it. That way, you can follow what I am doing as I share my screen with you. If you are lucky enough to have two screens, you can move one window to the second screen.

[show how to

- login to Jupyter server,
- get “Whirlwind Tour to Python”,
- open new Jupyter Notebook,
- open new Terminal and start Python interpreter,
- shut down running processes
- menu items in Jupyter Notebook tab.

]

How do you start?

We go the easy way (so you can build it up from there):

- Choose Python.
- Skip the installation bit: Jupyter Notebooks.
- HOWEVER: You have to type yourselves! Don't copy-paste (yet)![1]
- Have a play.

[1] If you try to copy-paste from these slides, you will sometimes get a syntax error because of the characters used in the PDF.

Hello World!

It is a tradition in the programming world that you start learning a new programming language by writing a program that prints out the words "Hello World!" to your standard output (for example, the terminal).

Hello World! (in C++)

helloworld.cpp:

```
#include <iostream>

using namespace std;

int main() {
   cout << "Hello World!\n";
   return 0;
}
```

compile:

```
g++ -o helloworld helloworld.cpp
```

execute:

```
./helloworld
```

Hello World! (in C++)

helloworld.cpp:

```cpp
#include <iostream>

using namespace std;

int main() {
  cout << "Hello World!\n";
  return 0;
}
```

compile:

```
g++ -o helloworld helloworld.cpp
```

execute:

```
./helloworld
```

Let's first have a look at how this would look like in C++, a compiled language.

This is the program: You include the necessary tools for inputs and outputs, and define a namespace, which means, a scope for your variable names. Don't worry about all these details now, if you go back to this slide at the end of the course, you will hopefully understand (most of) what's going on. You define your main function, print the words to your standard output, and return zero.

The next step, as you will remember, is compiling the program, which is what we are doing in the next box.

And finally, we execute the program, and it will print "Hello World!".

Don't be scared now, we will now do the equivalent in Python.

Hello World! (in Python)

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

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

That's it. A bit easier.
Let's do it all together now.
[Run Hello World, explain how to actually execute the program]
To make it a bit more interesting, let's play around with it together.
Don't worry if you don't understand why these commands look like they do, and what the syntax exactly means. We will go through this in a moment.

Just a short note on terminology, to make sure we are on the same page: "syntax" is the form of a computer program, i.e., how are things written down. On the other hand, "semantics" is the meaning of what is written down.

Hello World!

```
name = "Marion"

print("Hello " + name + "!")
```

Hello World!

```
name = "Marion"

print("Hello " + name + "!")
```

Let's see what happens if we do this. Please, again, type along as I type! [type and execute] We see that we can put the name into the output by defining it like shown here. If we change the name [do that and execute] we change the output.

Hello World!

```
name = "Marion"

print("Hello " + name + "!")

mySum = 2+3

print("I can add: 2+3 = ", mySum)
```

```
name = "Marion"
print("Hello " + name + "!")
mySum = 2+3
print("I can add: 2+3 = ", mySum)
```

Next, let's use numbers for this, what happen's? [type and execute] As you can see, it has actually added the numbers together. However, if you look closely at the syntax and compare with what we have done before, it has changed. What happens if we add the quotation marks? [demonstrate]

You can see, there is a lot going in these few lines. Let's stop for a moment and look into the details.

Basics

In this section we will look into the very basics of programming, to give you the foundations to start from. A lot of it will be transferrable to other programming languages, even if the exact syntax changes.

Basic Data Types

- Strings: "Heinz", 'Banana', 'He said "Hello"'
- Integers: 1, 2, 3, 22222222, -777
- Floats: -1.2, 0.0, 2.7182
- Booleans: True, False

Basic Data Types

- Strings: "Heinz", 'Banana', 'He said "Hello"'
- Integers: 1, 2, 3, 22222222, -777
- Floats: -1.2, 0.0, 2.7182
- Booleans: True, False

When you are programming, you have to be clear about what type each "item" you are handling has. Each data type has different rules. In the most basic version, Python has the following data types:

- Strings, which are bits of text, are noted by single or double quotation marks, and can be nested.
- Integers, which are whole numbers, positive or negative.
- Floats, of floating points numbers, which allow for a fractional part.
- Booleans, which are truth values, and can be either true or false.

Basics: Variables

- "I reserve a space in memory for my data bit, and I call it by the name x"
- Syntax: $name = value$

The next essential concept you need to know for programming in Python are variables.

By writing "name = value", you are basically saying "I reserve a space in memory for my data bit, which is value, and I call it name".

Then you can use the name to further process your data bit, as we have seen it in the example with the print statement.

Examples

```
print('He said "Hello"')

myString = 'He said "Hello"'

print(myString)

type(myString)
```

Examples

```
print('He said "Hello"')
myString = 'He said "Hello"'
print(myString)
type(myString)
```

In this example we see the print statement again. It will usually expect a string, or something that can be converted to a string.
Here we are saying that the variable “myString” should have the string value 'He said "Hello"', and then put the print statement separately. The “type” command is able to tell us, what type the variable “myString” has. [demonstrate]
As a side note, many programming languages, in particular compiled programming languages, require you to explicitly state the data type of your variable. We do not need to do that here.

Examples

```
print(2+5)

print("2+5 = " + 2 + 5)

print("2+5 = " + str(2 + 5))

print("2+5 = ", 2 + 5)
```

Examples

```
print(2+5)
print("2+5 = " + 2 + 5)
print("2+5 = " + str(2 + 5))
print("2+5 = ", 2 + 5)
```

As I said before, “print” expects a string or something that can be converted to a string. Here are some examples what happens if I am dealing with integers. Please try each of these examples for yourself, and see what the output is. One will give you an error. [let students try] In the first case, the interpreter recognises that I have to integers that want to be added together, and gives me the answer. The second case produces an error and says “TypeError: can only concatenate str (not "int") to str". The plus sign actually means two different things for numbers and for strings! You can “glue” two strings together using the plus sign, and you can add to numbers together, as we would expect.

Examples

```
print(2+5)
print("2+5 = " + 2 + 5)
print("2+5 = " + str(2 + 5))
print("2+5 = ", 2 + 5)
```

The third example shows you how to explicitely cast a number to a string. We tell it “add 2 and 5 together, make it a string, and concatenate it to the other string”.

To make things a bit more complicated (or easier, depending on your point of view), you can do the same things by simply putting a comma instead of the plus in the print statement. Ask me about the meaning behind that at the end of the course, when we have talked about functions!

Examples

```
type(true)

type(True)

1+True

type(1+True)

bool(True+False)

bool(True and False)

bool(True or False)

not(False)
```

Examples

```
type(true)
type(True)
1+True
type(1+True)
bool(True+False)
bool(True and False)
bool(True or False)
not(False)
```

Let's take a closer look at booleans. They seem a bit boring, but we will see later that they are really essential for programming. Try these commands in your interpreter. [let them try] You will have seen that the values are case sensitive, so you have to make sure to write "True" and "False" with capital letters. You will also have noticed that "True" has the numerical value one, and "False" is zero. You can invert the value with "not", and combine two values logically with "and" and "or".

Basic Operations

- String: concatenation with $+$
- Bool: and, or, not
- Numerical data: $+$, -, *, /, %, **, abs, ...
- Order of execution:
  1. ()
  2. **
  3. *,/
  4. $+$, -
  5. Left-to-right (except exponentiation!)

$\Rightarrow$ Use parenthesis to make sure!

Basic Operations

- String: concatenation with +
- Bool: and, or, not
- Numerical data: +, -, *, /, %, **, abs, ...
- Order of execution:
  1. ()
  2. **
  3. *,/
  4. +, -
  5. Left-to-right (except exponentiation!)

⇒ Use parenthesis to make sure!

These are the basic operations we have for our data types: String concatenation with the plus sign, logical operations on booleans with and, or, and not, and for floats and integers we have addition, subtraction , multiplication, division, modulo (i.e., the remainder of a division), exponentiation, absolute value, and so on.

Basic Operations

- String: concatenation with +
- Bool: and, or, not
- Numerical data: +, -, *, /, %, **, abs, ...
- Order of execution:
  1. ()
  2. **
  3. *,/
  4. +, -
  5. Left-to-right (except exponentiation!)

⇒ Use parenthesis to make sure!

It is very important to be aware how these operations are executed in long arithmetic expressions. Generally, if you have something like a plus b plus c, it happens from left to right: First a plus b, and then plus c. However, if you have exponents, like a to the power of b to the power of c, the right-most operations happens first: b to the power of c, and then a to the power of the result. Exponents bind stronger than times and divided by, i.e. exponents are computed first, and times and divided by is stronger than plus and minus. If in doubt, just use parenthesis.

You will have the chance to play around with this in a moment, and find out for yourselves how it works.

Basics: Comments (and Documentation)

```
# This is my programme to demonstrate how to
# do simple calculations in Python.

myNumber = 2

myOtherNumber = myNumber+5

myNumber = myOtherNumber/2 # I have to divide
                           # by 2 here, as the
                           # results are
                           # otherwise rubbish

print(myNumber)
```

Basics: Comments (and Documentation)

```
# This is my programme to demonstrate how to
# do simple calculations in Python.

myNumber = 2

myOtherNumber = myNumber+5

myNumber = myOtherNumber/2 # I have to divide
                           # by 2 here, as the
                           # results are
                           # otherwise rubbish

print(myNumber)
```

But first, I want to show you two more things.

One is comments. When you are writing your code, your intentions and your thinking might seem obvious, and you just want to get it running. However, believe me: If someone else is looking at your code at some point, and this someone else might be you in two weeks time, things are very probably not that obvious anymore. I found myself puzzling over my own code and why on earth I have done things like that quite often.

In order to put a comment in your code you have to tell the interpreted that this bit is just for the humans reading the code, not interesting for the computer. In Python this is done by putting a hash symbol in front of the comment. This can be either at the beginning or in the middle of a line.

Debugging and Types of Errors

- Errors in computer programs are called “bugs” for historic reasons.
- For complex projects, you will usually spend more time testing and debugging than writing code.
- Three types of errors:
  - Syntax errors - written the code wrongly
  - Semantic errors - written the wrong code
  - Runtime errors - something’s wrong with the code (during execution)

Debugging and Types of Errors

- Errors in computer programs are called "bugs" for historic reasons.
- For complex projects, you will usually spend more time testing and debugging than writing code.
- Three types of errors:
  - Syntax errors - written the code wrongly
  - Semantic errors - written the wrong code
  - Runtime errors - something's wrong with the code (during execution)

The other thing I want to mention at this points are errors. As soon as you start programming you will make mistakes. Everyone does that, even the most experienced programmers spend a lot of time hunting for bugs in their code. "Bugs" is a common term for errors in computer programs. It is said that this term originates from a time when actual insects in the hardware of a computer or another machine were the cause for malfunctions. There is a number of different opinions on when the term was first used, but I won't go into that here. Looking for and removing these errors is also called "debugging".

Debugging and Types of Errors

- Errors in computer programs are called "bugs" for historic reasons.
- For complex projects, you will usually spend more time testing and debugging than writing code.
- Three types of errors:
  - Syntax errors - written the code wrongly
  - Semantic errors - written the wrong code
  - Runtime errors - something's wrong with the code (during execution)

Programming errors can broadly be divided into three types:

- Syntax errors are usually the easiest to spot. We have seen them before, when we wrote some code that was not valid, made a typo or so. Your interpreter or compiler will tell you straight away that something is wrong.

- Semantic errors are much harder to find. It means that the code that you have written does not do what you intended it to do. Because you have written it, you will probably have assumed that it does the right thing, so you have to question your thinking. Often it helps to explain your code to someone else, so that either they or yourself might spot the mistake.

Debugging and Types of Errors

- Errors in computer programs are called "bugs" for historic reasons.
- For complex projects, you will usually spend more time testing and debugging than writing code.
- Three types of errors:
  - Syntax errors - written the code wrongly
  - Semantic errors - written the wrong code
  - Runtime errors - something's wrong with the code (during execution)

- The third type is runtime errors. These will just appear during execution, for example because your code cannot cope with a certain type of input. An example would be a code that divides a number by another number given by user input, which errors if the user input is zero or a string.

Have a play!

You could try

- what happens if you add a float and an integer,
- what happens if you mix numbers and bools in arithmetic expressions,
- how setting parenthesis changes the result of a large arithmetic expression,
- to print statements that include variables of different data types,
- try to reproduce each of the error types,
- ...

Have a play!

You could try

- what happens if you add a float and an integer,
- what happens if you mix numbers and bools in arithmetic expressions,
- how setting parenthesis changes the result of a large arithmetic expression,
- to print statements that include variables of different data types,
- try to reproduce each of the error types,
- ...

Now it's time for you to try the things that we have learned about yourselves. I want you to think about data types, variables, the different operations, comments and errors, and find out more about how they work by experimenting. Write little bits of code and see what works, what doesn't, and how the output changes.

I wrote down some examples here, in case you are unsure what to do.

Again, if you have any question or need help, we are here to help.

I'll give you 15 minutes, then we'll continue. [set counter to 15m – https://timer.onlineclock.net/]

Getting Data in and out

Now that we have covered the basics, let's move on to the next level. I expect that most of you are interested in processing data in one way or the other. Most times, we do not want to explicitely write our data into our code, which is also called "hardcoding", and has a bad reputation.

Instead, we want to be able to provide data during runtime, and have a general program that is able to process the data.

How then do we get the data in and out of our program?

User Input

```
# Get some user input

x = input()

print(x)

type(x)    # This will be a string
           # if you don't convert it

# Get the user's name

name = input("What's you name?")

print("Hello " + name)
```

User Input

```
# Get some user input

x = input()

print(x)

type(x)    # This will be a string
           # if you don't convert it

# Get the user's name

name = input("What's you name?")

print("Hello " + name)
```

As an equivalent to the print statement, there is the input statement. It just takes the input from the command line or terminal, and you can do with it whatever you like. For example, here I am putting it into a variable x, and then output it. You can also add a prompting message to the user to tell them what kind of input you expect. Note that, just as print outputs a string, any user input will be interpreted as of type string, regardless of whether it is made out of letters, numbers, or symbols.

This kind of user input is, of course, not suited for inputting large data sets. It is rather for getting in, for example, parameters, the path to the data set, or other settings or instructions to the program.

Reading and writing files

```
# Create a file object
myFile = open("testfile.txt", "w")
```

Two things to note here:

- My object "myFile" is different from my file "testfile"!
- There are different modes:
  - read: r
  - (over-)write: w
  - append: a
  - read+write: w+ or r+

Reading and writing files

```
# Create a file object
myFile = open("testfile.txt", "w")
```

Two things to note here:

- My object "myFile" is different from my file "testfile"!
- There are different modes:
  - read: r
  - (over-)write: w
  - append: a
  - read+write: w+ or r+

The actual data input will usually happen over a file. Follow me in creating a new file. [demonstrate] As you will see, there is now a new file called "tesfile.txt" in your working directory. [show] But that is not the only thing that happened. At the same time, a file object called "myFile" was created in your program. These are two different things. For example, if I rename or delete the file object, my file in the file system won't change. This might sound trivial, but I thought I mention it, as it could lead to confusion.

The w here stands for "write". If there is already a file with this name, we will now overwrite its contents. If we instead open the file with "a" for "append", we start writing after the existing content. We can also open the file in read-only mode, or in read-and-write mode.

Reading and writing files

```
# Create a file object
myFile = open("testfile.txt", "w")

# Write - note special characters!
myFile.write("This is some text. \n \
And some more.")
myFile.write("\n\nI can also add numbers \
like this: %d %d \n" %(22, 333))

myFile.write(str(222))

# Don't forget to close the file
myFile.close();
```

see also f-strings

Reading and writing files

```
# Create a file object
myFile = open("testfile.txt", "w")

# Write - note special characters!
myFile.write("This is some text. \n \
And some more.")
myFile.write("\n\nI can also add numbers \
like this: %d %d \n" %(22, 333))

myFile.write(str(222))

# Don't forget to close the file
myFile.close();
```

see also f-strings

Now we are writing some text and numbers into our file. Note that, again, everything needs to be a string. We can put in numbers either by converting them directly to a string, or by putting in a place holder, like this “percent d”, for numerial or decimal values, and then putting in the number after the string. The “backslash n” symbol stands for a line break.

The single backslash is a line continuation character and tells the interpreter that we are not done yet with that command, but that it continues in the next line.

At the end, we close our file, and then can no longer write to it unless we open it again.

Try for yourself to write something into the file you just created, and then check the content of your file!

[give them some minutes to do that]

Reading and writing files

```
# Create a file object (this time for reading)
myFile = open("testfile.txt", "r")

# Read it and print it to screen
print(myFile.read())

# Try this:
print(myFile.read(7))
print(myFile.readline())
print(myFile.readlines())

# Don't forget to close the file
myFile.close();
```

Reading and writing files

```
# Create a file object (this time for reading)
myFile = open("testfile.txt", "r")

# Read it and print it to screen
print(myFile.read())

# Try this:
print(myFile.read(7))
print(myFile.readline())
print(myFile.readlines())

# Don't forget to close the file
myFile.close();
```

In the next step, we will read out data from the file that we have just created. There are different ways of doing this. In this example, I combine the reading directly with a print statement, so that I see the result. Usually you would probably put the output of the reading operation into a variable, and continue to process it.

Have a go with these four different version of file reading. [give them some minutes]

You have seen that you can read the whole file in one go, read single characters from the file, or a whole line. If you apply the “readline” command repeatedly, it will go on to the next line of your file.

Interesting is the output of the “readlines” command. It gives you a structure that has all the lines of the file in it, as separate strings, divided by commas.

What do we have here?

```
myList = [1, 2, 3, 4, 5] # A list!

print(myList)

print(myList[3])  # Note: [] not ()

print(myList[0]) # Start with 0!

print(myList[-1]) # Go backwards

print(myList[1:4]) # Include first,
                   # exclude last

print(myList[:2]) # More slicing
```

What do we have here?

```
myList = [1, 2, 3, 4, 5] # A list!
print(myList)
print(myList[3])  # Note: [] not ()
print(myList[0]) # Start with 0!
print(myList[-1]) # Go backwards
print(myList[1:4]) # Include first,
                   # exclude last
print(myList[:2]) # More slicing
```

This kind of structure is called a list, and it is an essential data structure in Python programming.

We can access the entries using square brackets. The first entry in the list is entry number zero, because we are programmers now, and programmers and computer scientists like to start counting from zero. We can even go backwards, in a periodic sense, and land at the last entry of the list at -1, the second-last at -2, and so on. We can take a slice of the list, for example the entries one to three – note that, in this notation, the first entry given is included and the last excluded. If we just put a colon, we mean that we go all the way to that side, so here we go from the beginning of the list, which is entry zero, to entry 1. You can just put a colon to indicate you mean the whole list.

Have a Play!

https://www.w3schools.com/python/python_lists.asp

Have a Play!

https://www.w3schools.com/python/python_lists.asp

This is something that is worth getting your head around. Please go to the website given here, read through it and try their examples. At the end, you will find a list of methods on lists, with links to further information. Even though the website uses all toy examples, these are really useful tools in data processing. Try to think how you would use these methods on your actual research data.

I'll give you 10 minutes, then we'll continue. [set counter to 10m – https://timer.onlineclock.net/]

For Arithmetic Operations Better: Arrays

```
import array as arr

myList = [1, 2, 3, 4, 5]

myArray = arr.array('i', myList)
```

Alternative import:

```
from array import *

myList = [1, 2, 3, 4, 5]

myArray = array('i', myList)
```

see also numpy arrays, (here a tutorial)

For Arithmetic Operations Better: Arrays

```
import array as arr

myList = [1, 2, 3, 4, 5]

myArray = arr.array('i', myList)
```

Alternative import:

```
from array import *

myList = [1, 2, 3, 4, 5]

myArray = array('i', myList)
```

see also numpy arrays, (here a tutorial)

Just as a side note, if you are working with numerical data, you might prefer to use arrays. Arrays can perform arithmetic operations directly on a slice, or the whole array.

Here are two examples of how to convert lists of integers into integer arrays. I won't go into more details about that here.

Repetitions and Conditions

Now we know how to do simple computations in Python. But imagine a situation where you have read in all 1000 lines of your data file in a neat list, and now have to perform the same computation with all of them. Or you have 1000 files that you need to process all in the same way. Or you want to do repeatedly apply an operation to your data until it fulfils a certain condition. You definitely do not want to do this by hand, now that you know how to program!

While-Loop

```
mySum = 0

while mySum < 100:

    mySum = mySum + 3 # Mind the indentation!

print(mySum)
```

While-Loop

```
mySum = 0
while mySum < 100:
    mySum = mySum + 3 # Mind the indentation!
print(mySum)
```

This is a simple while-loop. The example does not make a lot of sense, it only adds up threes as long as the sum is smaller than 100, and then prints the sum. Let's do this one together – again, follow me in typing. [type and execute] Note: this will only print the sum once. If I now do this [add indentation before print statement and execute] it will print the sum after each step! The indentation tells the interpreter what belongs inside the while loop.

While-Loop

```
mySum = 0
while mySum < 100:
    mySum = mySum + 3 # Mind the indentation!
print(mySum)
```

If you mess up your indentation by using different numbers of spaces it will complain. But if you forget about this and put something into the wrong indentation level, as you might think it looks prettier or so, you will get into trouble, as the semantics of your whole program will change. Take this as a warning: Be mindful of indentations. Some other programming languages use brackets instead of indentations. Also: Don't forget the colon! That happens to me all the time.

While-Loop

```
mySum = 0
while mySum < 100:
    mySum = mySum + 3 # Mind the indentation!
print(mySum)
```

We can also now see why booleans are important. The expression "mySum $<$ 100" is either true or false, it is evaluated as a bool. If I would write "while true" my loop would just continue to run indefinitely. I can then tell it explicitely to stop, for example if a certain conditions applies, using the key word "break". If I write "while false", the interpreter will simply skip over the loop and not execute its body (which is how we call the inner part of a loop).

For-Loop

```
mySum = 0

for i in range(5): # i goes from 0 to 4

    mySum += i # mySum = mySum + i

print(mySum)
```

For-Loop

```
mySum = 0
for i in range(5): # i goes from 0 to 4
    mySum += i # mySum = mySum + i
print(mySum)
```

Another type of loop is a for-loop. This might look at bit complicated at first, but I'll explain it to you. The only line that really needs explanation is actually the second one in this example. What it does is that it defines a variables i – we could call it anything else, but as it is an integer in this example, it is customary to call it i – and we iterate through a list of values for i. It could be a list as we have seen it previously.

For-Loop

```
mySum = 0
for i in range(5): # i goes from 0 to 4
    mySum += i # mySum = mySum + i
print(mySum)
```

But I am showing you here a common way of creating a sequence of integers that run from zero up to a certain value, the “range” command. “range(5)” will give you a list with the numbers 0, 1, 2, 3, 4. You see, the list has 5 entries, and we start counting, again, from zero. [demonstrate what “range(5)” does, using “*range(5)” to unpack the range sequence]

You can also start from a different number than zero, and go in different steps than steps of one. The syntax is “range(start, stop, step)”. [demonstrate this]

For-Loop

```
mySum = 0

i = 0

mySum = mySum + i

i = 1

mySum = mySum + i

i = 2

mySum = mySum + i
⋮
```

For-Loop

```
mySum = 0
i = 0
mySum = mySum + i
i = 1
mySum = mySum + i
i = 2
mySum = mySum + i
⋮
```

This is what happens in the loop. I have detangled it here for you. You start with the variable mySum being zero, and then define your integer i and give it its first value, which is zero. Then you add it to the sum. Next, you move one further in the range and assign the next value, 1, to i, and add that to the sum. Then i gets the next value, which is 2 in this example, and so on.

For-Loop – Real-Life Examples

```
for i in range(1, 100):

  filename = "myfile"+str(i)+".dat"

  #Do stuff with that file
```

```
for i in range(100):

  currentFile = myFileList[i]

  #Do stuff with that file
```

```
for myFile in myFileList:
  #Do stuff with that file
```

For-Loop – Real-Life Examples

```
for i in range(1, 100):
  filename = "myfile"+str(i)+".dat"
  #Do stuff with that file
```

```
for i in range(100):
  currentFile = myFileList[i]
  #Do stuff with that file
```

```
for myFile in myFileList:
  #Do stuff with that file
```

Given the previous example you might wonder what this kind of loop is useful for, but as soon as you start programming you will find that you use it a lot. Practical examples would be looping over all the files in a directory, or all the files in a file list, as shown here. As I had mentioned before, you can directly loop over all elements in a list, so the second and third example are actually equivalent.

If-Statement

```
num1 = float(input("Give me a number!"))
num2 = float(input("Another number!"))

if num1 > num2:
  print("Your first number is bigger than \
        your second number.")
else:
  print("Your first number is not bigger \
        than your second number.")
```

(There will be some annoying whitespace if you type it like that, just remove the linebreaks and the whitespace in the messages in your code.)

If-Statement

```
num1 = float(input("Give me a number!"))
num2 = float(input("Another number!"))

if num1 > num2:
  print("Your first number is bigger than \
        your second number.")
else:
  print("Your first number is not bigger \
        than your second number.")
```

(There will be some annoying whitespace if you type it like that, just remove the linebreaks and the whitespace in the messages in your code.)

You will get to play with these things in a moment.

Back from real-world examples to toy examples. This example code takes two user inputs, converts the string that the input function sees to a floating point number, compares the numbers and gives an output depending on which one is bigger. Again, mind the indentation and the colon! The “else” part is optional, if we only want to make an output in one case, we just omit it.

As the footnote here says, the whitespace in the second line of each print statement is just for readability on this slide.

However, there is one big issue with this code.

Do we trust the user...?

```
try:
    num1 = float(input("Give me a number!"))
    num2 = float(input("Another number!"))
    if num1 > num2:
        print("Your first number is bigger than \
              your second number.")
    else:
        print("Your first number is not bigger \
              than your second number.")
except:
        print("This wasn't a valid input, \
          I'm afraid.")
```

```
try:
  num1 = float(input("Give me a number!"))
  num2 = float(input("Another number!"))
  if num1 > num2:
    print("Your first number is bigger than \
       your second number.")
  else:
    print("Your first number is not bigger \
       than your second number.")
except:
    print("This wasn't a valid input, \
      I'm afraid.")
```

Do we trust the user to only input numbers? What happens if they input strings or other symbols?

A simple way to deal with such critical code parts is a “try-except” statement. We put the code in the “try” part, and in the “except” part we define what is to be done should the critical case occur. This can be a simple error message, which is still better than just a program crash, or a way of actually dealing with the problem. For example, we could check whether the user has actually written out the number as a word.

Do we trust the user...?

```
try:
  num1 = float(input("Give me a number!"))
  num2 = float(input("Another number!"))
  if num1 > num2:
    print("Your first number is bigger than \
        your second number.")
  else:
    print("Your first number is not bigger \
        than your second number.")
except:
    print("This wasn't a valid input, \
      I'm afraid.")
```

You should, however, be aware that "try-except" is not always great if you are developing or debugging a code, as the interpreter will no longer tell you which line in the "try" bit has actually errored. It is, however, useful for user-facing code.

Have a play!

You could try

- writing numeric data into a file using a loop, and reading them into a list,
- defining a 2d list using a 'lists in a list' notation [[...],[...]] and try accessing its elements,
- doing nested loops,
- thinking about what type of errors *try-except* is able to catch (see slide 22),
- ...

Have a play!

You could try

- writing numeric data into a file using a loop, and reading them into a list,
- defining a 2d list using a 'lists in a list' notation [[...],[...]] and try accessing its elements,
- doing nested loops,
- thinking about what type of errors *try-except* is able to catch (see slide 22),
- ...

We have learned quite a lot of new concepts, and I would like you to have a play with all of this now. Again, here are some examples what you could do, but feel free to come up with your own examples, and don't hesitate to ask for help.

We will continue with the last part of this training in 20 minutes.

[set counter to 20m – https://timer.onlineclock.net/]

Functions

Let's move on to our last topic for today: functions. Imagine you have written a particularly useful bit of code, for example a code that finds all the occurences of a word in a text and counts them, or replaces them by something else. Or a code that does a very specific computation with your data. And now you want to do the same in a different context, with different words or data. What do you do? Do you copy-and-paste the code and replace the words or numbers that will change in the new situation? What if you then realise you have to change a critical part of this program, will you remember to do that in all the copies of your code?

Functions

We've seen functions (==useful code blocks that we need again and again, so we give them a name), for example *print*, *read(5)*, *open("testfile.txt", "w")*, etc.

$\Rightarrow$ Somewhere someone must have written some code for that...

Can we do that as well?

We've seen functions (==useful code blocks that we need again and again, so we give them a name), for example *print*, *read(5)*, *open("testfile.txt", "w")*, etc.

⇒ Somewhere someone must have written some code for that...

Can we do that as well?

This is where functions come into play. In fact, we have already used functions today. We have used “print”, “read”, “input” and “open”, given them parameters, and just accepted that this would perform more or less complex tasks. But of course, this is not some kind of computer magic, but somebody has written the code for what would happen if someone types the command “print” and gives it a string. We are just at the far end of a line of computer programmers that tell the computer what to do if somebody hacks on these keys on the keyboard in a certain order.

Functions

- Function Definition
- Arguments
- Return statement
- Function call
- Scope of variables

What do we need to write our own functions? We need to know how we define our function, and which arguments, or parameters, it accepts. We need to define whether the function is only supposed to do something, like the print statement take its arguments and prints it on the screen, or the write function writes text into a file. Or whether we want it to return a value that, for example, can be asigned to a variable, like the input function. It will also become important that we are aware of how we should name our functions and variables, and from where in our program we can access them.

Functions

```
def myFunction(parameter1, parameter2):
    mySum = parameter1 + parameter2
    return mySum

def anotherFunction(parameter):
    print("Here you go: ", parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

Functions

```
def myFunction(parameter1, parameter2):
  mySum = parameter1 + parameter2
  return mySum

def anotherFunction(parameter):
  print("Here you go: ", parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

Let's write our first little function. It is simply a function that adds two parameters. (One tip, a function for sums actually already exists in Python and is called, very sensibly, "sum".) We define it using the keyword "def", and again we use colons and indentation to show what belongs into the body of the function. In order to get our result out of the function we use the "return" keyword. We also define another function that does not have a return value, but just prints something. [type function definitions, show that nothing happens on execution] In order to actually execute the functions we have to call them. So, we assign the output of the first function to the variable "myVar", and then use the second function to print the result. [demonstrate] We could have skipped the bit with the variable and used the call of the first function as parameter of the second. [demonstrate]

Functions

```
def myFunction(parameter1, parameter2):
    mySum = parameter1 + parameter2
    return mySum

def anotherFunction(parameter):
    print("Here you go: ", parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

Variables defined inside a function are only available in that function.

Functions

```
def myFunction(parameter1, parameter2):
  mySum = parameter1 + parameter2
  return mySum

def anotherFunction(parameter):
  print("Here you go: ", parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

Variables defined inside a function are only available in that function.

As you can see here, we assign the result of the function call to a new variable with a different name, “myVar”. We could have chosen the same name, mySum, but what I want to make clear here is that variables have a certain scope, and are only known within this scope. If I try to access “mySum” from outside the function, without defining it here, the interpreter won’t know it. [demonstrate] I could have two variables with the same name in here, one in the function, one outside, and one could be an integer and the other a string. They wouldn’t know of each other.

Functions

```
def myFunction(parameter1, parameter2):
    mySum = parameter1 + parameter2
    return mySum

def anotherFunction(parameter):
    print("Here you go: ", parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

Variables defined inside a function are only available in that function.

We should really give the functions and variables better names!

Functions

```
def myFunction(parameter1, parameter2):
  mySum = parameter1 + parameter2
  return mySum

def anotherFunction(parameter):
  print("Here you go: ", parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

Variables defined inside a function are only available in that function.

We should really give the functions and variables better names!

Also, this code gives you a counterexample of good naming practice. If I look at the function calls I cannot guess at all, what the functions will do, or what the variable will contain. It is good practice that your function and variable names make it very clear what they are. Do not use cryptic abbreviations or names like “a, b c”, or “x1, x2, x3”. I have seen this very often, and the same applies as I have already said about comments: If somebody else looks at your code, or if you look at your code at a later point, things might not make sense anymore at all. In fact, if your variable and function names are meaningful you will need much less comments to make it understandable.

Flow of Execution (and debugging)

```
import pdb
pdb.set_trace()

def myFunction(parameter1, parameter2):
    mySum = parameter1 + parameter2
    return mySum

def anotherFunction(parameter):
    print(parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

```
import pdb
pdb.set_trace()

def myFunction(parameter1, parameter2):
  mySum = parameter1 + parameter2
  return mySum

def anotherFunction(parameter):
  print(parameter)

myVar = myFunction(22, 55)

anotherFunction(myVar)
```

[do this only if enough time] In order to show you in which order the interpreter steps through the code, and to give you a hint how to debug your code if it gets a bit complicated, I am showing you here how to invoke the Python debugger. As a default, Python comes with a certain toolset of functions. If we want to use a specific library, we have to import it like shown here. Importing a library makes an additional set of functions available for us. Here I say that, from the Python debugger library "pdb", I now want to use the "set_trace" function. That will stop the debugger exactly here. I then can use "s" and return to step through the code. [demonstrate–don't use Jupyter Notebook, but create file and use terminal!–and show where the variable name "mySum" is known and where not]

Advanced: Recursion

https://realpython.com/python-thinking-recursively/

https://realpython.com/python-thinking-recursively/

Don't worry, we will not talk about recursion today, I just thought I will put this link here in case any of you wants to have a look on advanced usage of functions later at home.

Have a play!

You could try

- thinking about how to best name the variables and functions in the examples above, and why meaningful names are crucial,
- writing your own functions, and let one be called from within the other,
- writing a code that uses all that you have learned today,
- use the Python debugger to step through your code,
- ...

Have a play!

You could try

- thinking about how to best name the variables and functions in the examples above, and why meaningful names are crucial,
- writing your own functions, and let one be called from within the other,
- writing a code that uses all that you have learned today,
- use the Python debugger to step through your code,
- ...

I'll give you another 20 minutes to put these things into practise, and then we'll have a short wrap-up before we are finished!

[set counter to 20m – https://timer.onlineclock.net/]

How to continue from here?

- Resources from the materials slide
- Research Methods Cafe
- Programmers are social!

That's it! You should now have a good understanding of the basics of programming. I have given you the links to further material already on one of the first slides. Of course there is always more to learn. If you start writing your own research code now and have any questions or problems, we offer drop-in sessions every Wednesday morning from 10.30 until 12. Currenly these are online using teams, the links here brings you to the website with information on how to join.

Also, despite all clichees, programmers are usually very social people and happy to help. There are lots of online forums where you can ask for help, or often even find the answer to your question by searching, as somebody else will likely have had the same question before you. If you are really into it, there are hackathons, or things like "code-wars" or "advent of code" where your can test your programming skills in coding challenges.

Before you leave this session, it would be great if you could leave feedback on this course in the chat. Please just write one positive thing and one thing that could be improved.

Thank you very much! I hope you found this course useful.