Python

Python (http://python.org) is a scriping language that we'll use throughout the course. For the coding part of lectures we will be using jupyter (https://jupyter.org/). In the slides you will find instructions to install both.

Variables

Let's start easy and create a text string, which we save to a variable called `greeting`, which we proceed to print.

In [ ]:

```
greeting = "Hello Dirk!"
print(greeting)
```

The `type` function returns the *type* of a variable.

In [ ]:

```
type(greeting)
```

Python is *strogly typed*, meaning that each variable has a *fixed* type, but **dynamic (or 'duck;) typing** allows to reassign `greeting` to an integer. C.f. Wikipedia on strong vs. weak typing (https://en.wikipedia.org/wiki/Strong_and_weak_typing).

In [ ]:

```
greeting = 42
```

In [ ]:

```
greeting # don't need the print statement in jupyter notebooks
```

In [ ]:

```
type(greeting)
```

Getting help

`help(...)` prints a help message to most things.

In [ ]:

```
help(int)
```

In [ ]:

```
?int # in jupyter notebooks the ? also works
```

In [ ]:

```
type(4.2)
# foating-point numbers contain a dot, 1. or .5 also work
```

Boolean values

The boolean types are `False` and `True`.

In [ ]:

```
type(False), type(True)
```

Lists

Lists are created with square backets and can contain most python objects.

In [ ]:

```
my_list = [1, 'String', 3.4, []] # [] = empty list
```

In [ ]:

```
list(range(10))
# built-in funciton to create a list
# here: from 0 (inclusive) to 10 (exclusive)
```

In [ ]:

```
list(range(3, 15, 3)) # from 3 to (and excluding) 15, in steps of 3
```

In [ ]:

```
my_list[0], my_list[-1] # first and last element
```

In [ ]:

```
my_list[1:3] # second to third (exclusive)
```

In [ ]:

```
my_list[1:] # second to the end
```

In [ ]:

```
my_list[:-1] # first to last (exclusive)
```

In [ ]:

```
my_list[::-1] # reversed, makes a copy
```

List comprehensions

List comprehensions let you create lists on-the-fly. This is very powerful and (to the trained eye) easy to read. But don't overdo!

We start with `range(10)` (as we've used above), take each element (`i`), square it (`i**2`) and add one.

In [ ]:

```
[i**2 + 1 for i in range(10)]
```

In [ ]:

```
[i**2 + 1 for i in range(10) if i % 2 == 0]
# if filters the numbers, here: only even ones
```

In [ ]:

```
print(i) # i would be still alive in python2 ..., this is a common gotcha
```

In [ ]:

```
my_list.append(2) # append an element
```

In [ ]:

```
my_list # list now contains an additional 2
```

In [ ]:

```
my_list.pop() # remove and return the last element, very useful
```

In [ ]:

```
my_list # the .pop() modified the list
```

In [ ]:

```
my_list.extend([2,'a']) # append another list
```

In [ ]:

```
my_list
```

In [ ]:

```
my_list.pop() # remove the appended elements
my_list.pop()
```

In [ ]:

```
my_list + [2, 'a'] # also appends, but doesn't modify my_list
```

In [ ]:

```
my_list
```

In [ ]:

```
my_list += [2, 'a'] # synonymous with .extend(...)
```

In [ ]:

```
my_list
```

In [ ]:

```
my_list += [2] # add another 2
my_list
```

In [ ]:

```
my_list.count(2) # count occurences of the element '2'
```

In [ ]:

```
my_list.index(3.4) # get the index of the element '3.4'
```

In [ ]:

```
my_list[2]
```

In [ ]:

```
my_list.index(2) # get the index of the _first_ '2'
```

In [ ]:

```
my_list.remove(2) # remove the _first_ 2
```

In [ ]:

```
my_list # the other 2s are still alive and well
```

In [ ]:

```
my_list.insert(0,2)
# insert another 2 ('cause we like 2s) at the beginning (index 0)
```

In [ ]:

```
my_list
```

In [ ]:

```
my_list.reverse() # in-place
```

In [ ]:

```
my_list
```

In [ ]:

```
my_list[::-1] # copy, as mentioned above
```

In [ ]:

```
letters = list("aAbBCc") # make a list out of a string
```

In [ ]:

```
letters
```

In [ ]:

```
letters.sort() # sort, in-place, use sorted(letters) to make a copy
letters
```

In [ ]:

```
letters.sort(key=str.lower) # use a function for generating sort keys
```

In [ ]:

```
letters
```

In [ ]:

```
# just to avoid confusion, str.lower is a normal funciton
str.lower('ALL CAPS!')
```

Tuples

Tuples are very similar to lists. Initialized with parenthesis instead of suqare backets, they share a lot of the properties of lists, only that they are not modifyable.

In [ ]:

```
my_tuple = ('a', 32, int)
```

In [ ]:

```
my_tuple[1]
```

In [ ]:

```
my_tuple[1] += 1
```

You can, however *expand* a tuple.

In [ ]:

```
my_tuple += (1,2)
```

In [ ]:

```
my_tuple
```

In [ ]:

```
my_tuple.pop() # ... but you can't remove elements
```

Strings

Strings of characters are used to represent text and share a lot of the properties of list. I'd encourage you to read the documentation (https://docs.python.org/2/library/stdtypes.html#string-methods).

In [ ]:

```
"" # empty string
```

In [ ]:

```
string_var = ",".join(("foo ", "Bar", "!")) # combine strings
```

In [ ]:

```
print(string_var)
```

In [ ]:

```
string_var.split() # split strings, by default at space ...
```

In [ ]:

```
string_var.split(",") # ... but you can choose another split character.
```

In [ ]:

```
"This is a nice day!"[::-1] # slicing works just as for lists
```

The `in` operator

`in` can be used to test membership in strings, lists, tuples, and other container types.

In [ ]:

```
"day" in "This is a nice day!"
```

In [ ]:

```
2 in range(10)
```

String formating

The `.format` function on strings lets you include variables into strings. It is very powerful, I'd again encourage you to read the documentation (https://docs.python.org/2/library/stdtypes.html#string-formatting).

In [ ]:

```
'Hello {0}, my name is {1}!'.format('friend', 'Dave')
```

In [ ]:

```
elements = ['H', 'He', 'Li', 'Be', 'B', 'C']
'The fifths element is {el[5]}'.format(el=elements)
```

In [ ]:

```
'{:*^12.5g}'.format(1.2345677)
```

Control flow

Control flow elements let you choose which parts of the code are executed when, possibly multiple times.

For loops

All loops and control statements in python rely on **indentation**. I will ususally use ipython's standard, 4 spaces, but you can use less. Everything following a colon (:) at the same level of indentation will belong to the preceding control statement.

The loop variable, whose name follows the `for` keyword, takes sequentially all elements of the collection whose name follows the `in` keyword, like so:

In [ ]:

```
# print all the elements in my_list
for element in my_list:
    # evertything here belongs to the for loop
    print(element)
# no indentation: end of loop
print("End of loop")
```

In [ ]:

```
for element in my_list:
    print(element)
    # all the code with indentation belongs to the for loop
    if type(element) == float:
        # all the code with this level of indentation
        # belongs to the if statement
        break # exit the loop here
```

Advice for beginners

Python's `for-else` can be confusing for beginners in programming. You can skip this section.

In [ ]:

```
for element in my_list:
    print(element)
    if type(element) == object:
        break
else:
    # this will be executed when the loop
    # exits with a break statement
    print("no object found")
```

In [ ]:

```
for element in my_list:
    print(element)
    if type(element) == list:
        break
else:
    # this won't be executed since my_list
    # contains another list
    print("no list found")
```

If-Elif-Else

If statements look sytactically similar to for-loops. We use indentation again to separate the lines of code belonging to the clauses.

In [ ]:

```
if "foo" in my_list:
    print("list has foo!")
elif "Foo" in my_list:
    print("list has big Foo!")
else:
    print("list has no foo!")
```

While

The last control statement is the while loop. On each iteration it tests a boolean condition until this condition is `False`. It's working like so:

In [ ]:

```
counter = 10 # pre-initialize
while counter > 0: # test if zero
    counter -= 1 # subtract one
    print(counter)
```

In [ ]:

```
elements  = [1,2,3,4]
while elements: # test for empty list
    if elements.pop() == 4: # remove element
        break
else:
    print("4 not found")
```

Concerning `while`

It is easy to mess up while loops and having them run eternally (and crashing your program). Use them sparingly and make extra sure your condition will end up being `False` ultimately (unless you write a daemon).

Reading data

We'll now read some data downloaded from google trends (https://www.google.com/trends/).

In [ ]:

```
# files are read using the function open
trends = open('data/trends.csv')
```

In [ ]:

```
trends.readline() # the first line contains a header
```

In [ ]:

```
trends.readline() # ... then follows data
```

In [ ]:

```
# we must remember to close the file
trends.close()
```

The `with` statement

To make our life easier, protect against missing files, closing errors, etc., Python has the `with` statement which adds some security for us. Let's use it to print the first 5 lines.

In [ ]:

```
# non-expert version
lines_to_print = 5
with open('data/trends.csv') as trends:
    for line in trends:
        print(line)
        lines_to_print -= 1
        if lines_to_print == 0:
            break
```

In [ ]:

```
# expert version
import itertools
with open('data/trends.csv') as trends:
    for line in itertools.islice(trends, 0, 5):
        print(line,end='')
```

On `import` and aliases

Python has a lot of built-in and external modules. They can be pulled into any script and jupyter session with the `import` keyword.

In [ ]:

```
import itertools as itls # make an alias
```

In [ ]:

```
itls == itertools # same thing
```

In [ ]:

```
?itls # get help, it's a *very* useful package
# (for experts), so read it if you have the time
```

In [ ]:

```
from itertools import islice, count # import individual functions
```

In [ ]:

```
islice == itertools.islice # same thing
```

In [ ]:

```
from datetime import datetime # date and time, read the docs!
```

Defining functions

Functions are defined using the `def` keyword, followed by the *name* of the fucntion and one or more arguments in parenthesis, like this:

In [ ]:

```
def ymd(date_string):
    """Convert a date string formated
       year - month - date
    to a datetime object."""
    format_string = "%Y-%m-%d"
    return datetime.strptime(date_string, format_string)
```