Lab 1: Python basics

Student I: thiqu264 (Thijs Quast)

Student II: lensc874 (Lennart Schilling)

A word of caution

There are currently two versions of Python in common use, Python 2 and Python 3, which are not 100% compatible. Python 2 is slowly being phased out but has a large enough install base to still be relevant. This course uses the more modern Python 3 but while searching for help online it is not uncommon to find help for Python 2. Especially older posts on sources such as Stack Exchange might refer to Python 2 as simply "Python". This should not cause any serious problems but keep it in mind whenever googling. With regards to this lab, the largest differences are how `print` works and the best practice recommendations for string formatting.

References to R

Most students taking this course who are not already familiar with Python will probably have some experience of the R programming language. For this reason, there will be intermittent references to R throughout this lab. For those of you with a background in R (or MATLAB/Octave, or Julia) the most important thing to remember is that indexing starts at 0, not at 1.

Recommended Reading

This course is not built on any specific source and no specific litterature is required. However, for those who prefer to have a printed reference book, we recommended the books by Mark Lutz:

- Learning Python by Mark Lutz, 5th edition, O'Reilly. Recommended for those who have no experience of Python. This book is called LP in the text below.
- Programming Python by Mark Lutz, 4th edition, O'Reilly. Recommended for those who have some experience with Python, it generally covers more advanced topics than what is included in this course but gives you a chance to dig a bit deeper if you're already comfortable with the basics. This book is called PP in the text.

For the student interested in Python as a language, it is worth mentioning

- Fluent Python by Luciano Ramalho (also O'Reilly). Note that it is - at the time of writing - still in its first edition, from 2015. Thus newer features will be missing.

A note about notebooks

When using this notebook, you can enter python code in the empty cells, then press ctrl-enter. The code in the cell is executed and if any output occurs it will be displayed below the square. Code executed in this manner will use the same environment regardless of where in the notebook document it is placed. This means that variables and functions assigned values in one cell will thereafter be accessible from all other cells in your notebook session.

Note that the programming environments described in section 1 of LP is not applicable when you run python in this notebook.

A note about the structure of this lab

This lab will contain tasks of varying difficulty. There might be cases when the solution seems too simple to be true (in retrospect), and cases where you have seen similar material elsewhere in the course. Don't be fooled by this. In many cases, the task might just serve to remind us of things that are worthwhile to check out, or to find out how to use a specific method.

We will be returning to, and using, several of the concepts in this lab.

1. Strings and string handling

The primary datatype for storing raw text in Python is the string. Note that there is no character datatype, only strings of length 1. This can be compared to how there are no atomic numbers in R, only vectors of length 1. A reference to the string datatype can be found **here (https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str)**.

[Litterature: LP: Part II, especially Chapter 4, 7.]

a) Define the variable `parrot` as the string containing the sentence *It is dead, that is what is wrong with it. This is an ex-"Parrot"*..

[Note: If you have been programming in a language such as C or Java, you might be a bit confused about the term "define". Different languages use different terms when creating variables, such as "define", "declare", "initialize", etc. with slightly different meanings. In statically typed languages such as C or Java, declaring a variable creates a name connected to a container which can contain data of a specific type, but does not put a value in that container. Initialization is then the act of putting an initial value in such a container. Defining a variable is often used as a synonym to declaring a variable in statically typed languages but as Python is dynamically typed, i.e. variables can contain values of any type, there is no need to declare variables before initializing them. Thus, defining a variable in python entails simply assigning a value to a new name, at which point the variable is both declared and initialized. This works exactly as in R.]

In [4]:

```
arrot = 'It is dead, that is what is wrong with it. This is an ex-"Parrot".'
arrot
```

Out[4]:

```
'It is dead, that is what is wrong with it. This is an ex-"Parrot".'
```

b) What methods does the string now called `parrot` (or indeed any string) seem to support? Write Python commands below to find out.

In [5]:

```
ir(parrot)
```

Out[5]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
```

```
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

c) Count the number of characters (letters, blank space, commas, periods etc) in the sentence.

In [6]:

```
len(parrot)
```

Out[6]:

66

e) If we type `parrot + parrot` , should it change the string itself, or merely produce a new string? How would you test your intuition? Write expressions below.

In [7]:

```
"""Functions behave differently for different types of variables.
he + function in Python concetenates strings. Therefore it will
how the string of text twice, as one string."""

arrot + parrot
```

Out[7]:

```
'It is dead, that is what is wrong with it. This is an ex-"Parrot".It is d
 ad, that is what is wrong with it. This is an ex-"Parrot".'
```

f) Separate the sentence into a list of words (possibly including separators) using a built-in method. Call the list `parrot_words` .

In [8]:

```
arrot_words = parrot.split()
arrot_words
```

Out[8]:

```
['It',
 'is',
 'dead,',
 'that',
 'is',
 'what',
 'is',
 'wrong',
 'with',
 'it.',
 'This',
 'is',
 'an',
 'ex-"Parrot".']
```

e) Merge (concatenate) `parrot_words` into a string again.

In [9]:

```
" ".join(parrot_words)
```

Out[9]:

```
'It is dead, that is what is wrong with it. This is an ex-"Parrot".'
```

2. Iteration, sequences and string formatting

Loops are not as painfully slow in Python as they are in R and thus, not as critical to avoid. However, for many use cases, *comprehensions*, like *list comprehensions* or *dict comprehensions* are faster. In this assignment we will see both traditional loop constructs and comprehensions. For an introduction to comprehensions, **this (https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html)** might be a good place to start.

It should also be noted that what Python calls lists are unnamed sequences. As in R, a Python list can contain elements of many types, however, these can only be accessed by indexing or sequence, not by name as in R.

a) Write a `for` -loop that produces the following output on the screen:

```
The next number in the loop is 5
The next number in the loop is 6
...
The next number in the loop is 10
```

[Hint: the `range` function has more than one argument.]
[Literature: For the range construct see LP part II chapter 4 (p.112).]

In [10]:

```
nput = range(5,11)

or i in input:
   a = "The next number in the range is {}".format(i)
   print(a)
```

```
The next number in the range is 5
The next number in the range is 6
The next number in the range is 7
The next number in the range is 8
The next number in the range is 9
The next number in the range is 10
```

b) Write a `for` -loop that for a given `n` sets `first_n_squared` to the sum of squares of the first `n` numbers (0..n-1).

In [15]:

```
 = 100  # If we change this and run the code, the value of first_n_squared should chan
e afterwards!
 your code goes here
irst_n_squared = []

or i in range(0,n):
   first_n_squared.append(i**2)
irst_n_squared = sum(first_n_squared)


irst_n_squared # should return 0^2 + 1^2 + ... + 99^2 = 328350 if n = 100
```

Out[15]:

```
328350
```

c) Write a code snippet that counts the number of **letters** in `parrot` (as defined above). Use a `for` loop.

In [18]:

```
ounter = 0
or s in parrot.lower():
   if s in set('abcdefghijklmnopqrstuvwxyz'):
       counter += 1
rint(counter)
```

```
47
```

d) Write a for-loop that iterates over the list `names` below and presents them on the screen in the following fashion:

```
 The name Tesco is nice
...
 The name Zeno is nice
```

Use Python's string formatting capabilities (the `format` function in the string class) to solve the problem.

[Warning: The best practices for how to do string formatting differs from Python 2 and 3, make sure you use the Python 3 approach.]
[Literature: String formatting is covered in LP part II chapter 7.]

In [19]:

```
ames = ['Tesco', 'Forex', 'Alonzo', 'Zeno']

or i in names:
   print("The name {} is nice".format(i))
```

```
The name Tesco is nice
The name Forex is nice
The name Alonzo is nice
The name Zeno is nice
```

e) Write a for-loop that iterates over the list `names` and produces the list `n_letters` (`[5,5,6,4]`) with the length of each name.

In [20]:

```
_letters = [0, 0, 0, 0]
or j in range(0,4):
           n_letters[j] = len(names[j])
rint(n_letters)
```

```
[5, 5, 6, 4]
```

f) How would you - in a Python interpreter/REPL or in this Notebook - retrieve the help for the built-in function `max` ?

In [21]:

```
elp(max)
ax?
```

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

g) Show an example of how `max` can be used with an iterable of your choice.

In [22]:

```
nput_numbers = [1,2,5,0,0,8,0,3]
ax(input_numbers)
```

Out[22]:

```
8
```

h) Use a comprehension (or generator) to calculate the sum 0^2 + ... + (n-1)^2 as above.

In [23]:

```
 = 100
irst_n_squared = sum([x*x for x in range(0, n)])
irst_n_squared
```

Out[23]:

```
328350
```

i) Solve assignment e) using a list comprehension.

[Literature: Comprehensions are covered in LP part II chapter 4.]

In [24]:

```
_letters = [0, 0, 0, 0]
_letters = [len(j) for j in names]
_letters
```

Out[24]:

```
[5, 5, 6, 4]
```

j) Use a list comprehension to produce a list `short_long` that indicates if the name (in the list `names`) has more than four letters. The answer should be `['long', 'long', 'long', 'short']` .

In [26]:

```
hort_long = ['long' if len(j) > 4 else 'short' for j in names]
hort_long
```

Out[26]:

```
['long', 'long', 'long', 'short']
```

k) Use a comprehension to count the number of letters in `parrot`. You may not use a `for`-loop. (The comprehension will contain the word `for`, but it isn't a `for ... in ...:`-statement.)

In [27]:

```
arrot_lowercase = parrot.lower()
umber_of_letters = sum([1 for s in parrot_lowercase if s in set('abcdefghijklmnopqrstu
wxyz')])
umber_of_letters
```

Out[27]:

47

[Note: this is fairly similar to the long/short task, but note how we access member functions of the values.]

l) Below we have the string `datadump`. Retrieve the substring string starting at character 27 and ending at character 34 by means of slicing.

In [29]:

```
atadump = "The name of the game is <b>old html</b>. That is <b>so cool</b>."
atadump[27:35]
```

Out[29]:

```
'old html'
```

l) Write a loop that uses indices to **simultaneously** loop over the lists `names` and `short_long` to write the following to the screen:

```
 The name Tesco is a long name
...
 The name Zeno is a short name
```

In [30]:

```
or person, length in zip(names, short_long):
   print("The name {} is a {} name.".format(person, length))
```

```
The name Tesco is a long name.
The name Forex is a long name.
The name Alonzo is a long name.
The name Zeno is a short name.
```

Note: this is a common programming pattern, though not particularly Pythonic in this use case. We do however need to know how to use indices in lists to work properly with Python.

m) Do the task above once more, but this time without the use of indices.

In [32]:

```
or name, length, in zip(names, short_long):
   print("The name {} is a {} name.".format(name,length))
```

```
The name Tesco is a long name.
The name Forex is a long name.
The name Alonzo is a long name.
The name Zeno is a short name.
```

[Hint: Use the `zip` function.]
[Literature: zip usage with dictionary is found in LP part II chapter 8 and dictionary comprehensions in the same place.]

n) Among the built-in datatypes, it is also worth mentioning the tuple. Construct two tuples, `one` containing the number one and `two` containing the number 1 and the number 2. What happens if you add them? Name some method that a list with similar content (such as `two_list` below) would support, that `two` doesn't and explain why this makes sense.

In [36]:

```
ne = (1,)    # Change this.
wo = (1,2)    # Change this
wo_list = [1, 2]
wo_tuple = one + two

"""When one wants to change values of a list, Python will allow this,
owever, when one wants to reassign values of a tuple, Python will not allow this.
lease run the code below to see. This makes sense, because there could be instances in
hich
ertain variables, in this case, tuples have to remain fixed throughout an entire proje
t. You don't want someone to
lter these variables."""

two_tuple[1] = 3
two_list[1] = 3
```

Out[36]:

```
[1, 2]
```

3. Conditionals, logic and while loops

a) Below we have an integer called `n`. Write code that prints "It's even!" if it is even, and "It's odd!" if it's not.

In [37]:

```
 = 7 # Change this to other values and run your code to test.

f n%2 == 0:
   print("It's even!")
lse:
   print("It's odd!")
```

```
It's odd!
```

b) Below we have the list `options`. Write code (including an `if` statement) that ensures that the boolean variable `OPTIMIZE` is True *if and only if* the list contains the string `--optimize` (exactly like that).

In [38]:

```
PTIMIZE = None       # Or some value which we are unsure of.
ptions = ['--print-results', '--optimize', '-x']  # This might have been generated by
a GUI or command line option

f any('--optimize' in i for i in options):
   OPTIMIZE = True
lse:
   OPTIMIZE = False

PTIMIZE
 Here OPTIMIZE should be True if and only if we found '--optimize' in the list.
```

Out[38]:

```
True
```

Note: It might be tempting to use a `for` loop. In this case, we will not be needing this, and you may *not* use it. Python has some useful built-ins to test for membership.

You may use an `else`-free `if` statement if you like.

c) Redo the task above, but now consider the case where the boolean `OPTIMIZE` is True *if and only if* the `options` list contains either `--optimize` or `-o` (or both).

In [42]:

```
PTIMIZE = None       # Or some value which we are unsure of.
ptions = ['x', 'y', '-x']  # This might have been generated by a GUI or command line o
tion


f any('--optimize' in i for i in options):
   OPTIMIZE = True
lif any('-o' in i for i in options):
   OPTIMIZE = True
lse:
   OPTIMIZE = False

 Here OPTIMIZE should be True if and only if we found '--optimize' or '-o' in the lis
.
PTIMIZE
```

Out[42]:

```
False
```

[Hint: Don't forget to test your code with different versions of the options list!

If you find something that seems strange, you might want to check what the value of the *condition itself* is.]

[Note: This extension of the task is included as it includes a common source of hard-to-spot bugs.]

d) Write out *a few* good tests that should (in an *extremely* informal sense) illustrate the correctness of your solution. Make sure that you actually try them out with your code.

Also make sure to write what the cases actually tell you, and why they are useful inputs to your set of tests. If you already have the test `options = ["hey"]`, adding the test `options = ["hello"]` doesn't add any possible ways of failing (or any coverage).

In [43]:

```
  Test 0: if options is the list below, OPTIMIZE should be True after.
  This test demonstrates that ...
 ptions0 =  ['a', 'b', '-x']
 ptions1 = ['--optimize', 'b', '-x']
 ptions2 = ['a', '-o', '-x']
 ptions3 = ['--optimize', '-o', '--optimize', 'o']

"""One can test that the solution works by changing the variable to
 ither options0, options1, options2, or options3
"""

 f any('--optimize' in i for i in options0):
    OPTIMIZE = True
 lif any('-o' in i for i in options0):
    OPTIMIZE = True
 lse:
    OPTIMIZE = False

  Here OPTIMIZE should be True if and only if we found '--optimize' or '-o' in the lis
 .
 PTIMIZE
```

Out[43]:

```
False
```

[Note: This way of testing your code is very primitive, but it's good to get used to constructing test cases. We will be discussing procedures and functions in the next lab.]

d) Sometimes we can avoid using an `if` statement altogether. The task above is a prime example of this (and was introduced to get some practice with the `if` statement). Solve the task above in a one-liner without resorting to an `if` statement. (You may use an `if` expression, but you don't have to.)

In [44]:

```
 PTIMIZE = None
 ptions = ['--print-results', '-o', '-x']  # This might have been generated by a GUI or
 ommand line option

 PTIMIZE = "--optimize" in options or "-o" in options# Replace None with your single li
 e of code.

  Here OPTIMIZE should be True if and only if we found '--optimize' or '-o' in the lis
 .
 PTIMIZE
```

Out[44]:

```
True
```

[Hint: What should the value of the condition be when you enter the then-branch of the `if` ? When you enter the else-branch?]

e) Write a `while` -loop that repeatedly generates a random number from a uniform distribution over the interval [0,1], and prints the sentence 'The random number is smaller than 0.9' on the screen until the generated random number is greater than 0.9.

[Hint: Python has a `random` module with basic random number generators.]

[Literature: Introduction to the Random module can be found in LP part III chapter 5 (Numeric Types). Importing modules is introduced in part I chapter 3 and covered in depth in part IV.]

In [54]:

```
mport random

hile random.uniform(0, 1) < 0.9:
   print("The random number is smaller than 0.9")
```

```
The random number is smaller than 0.9
The random number is smaller than 0.9
The random number is smaller than 0.9
The random number is smaller than 0.9
The random number is smaller than 0.9
The random number is smaller than 0.9
The random number is smaller than 0.9
The random number is smaller than 0.9
```

4. Dictionaries

Dictionaries are association tables, or maps, connecting a key to a value. For instance a name represented by a string as key with a number representing some attribute as a value. Dictionaries can themselves be values in other dictionaries, creating nested or hierarchical data structures. This is similar to named lists in R but keys in Python dictionaries can be more complex than just strings.

[Literature: Dictionaries are found in LP section II chapter 4.]

a) Make a dictionary named `amadeus` containing the information that the student Amadeus is a male, scored 8 on the Algebra exam and 13 on the History exam. The dictionary should NOT include a name entry.

In [55]:

```
madeus = {"Gender": "male", "Algebra": 8, "History": 13}
madeus
```

Out[55]:

```
{'Algebra': 8, 'Gender': 'male', 'History': 13}
```

b) Make three more dictionaries, one for each of the students: Rosa, Mona and Ludwig, from the information in the following table:

Name	Gender	Algebra	History
Rosa	Female	19	22
Mona	Female	6	27
Ludwig	Other	12	18

In [56]:

```
osa = {'Gender': 'Female', 'Algebra': 19, 'History': 22}
ona = {'Gender': 'Female', 'Algebra': 6, 'History': 27}
udwig = {'Gender': 'Other', 'Algebra': 12, 'History': 18}
```

c) Combine the four students in a dictionary named `students` such that a user of your dictionary can type `students['Amadeus']['History']` to retrive Amadeus score on the history test.

[HINT: The values in a dictionary can be dictionaries.]

In [57]:

```
tudents = {'Amadeus': amadeus, 'Rosa': rosa, 'Mona': mona, 'Ludwig': ludwig}
tudents['Amadeus']['History']
```

Out[57]:

```
13
```

d) Add the new male student Karl to the dictionary `students` . Karl scored 14 on the Algebra exam and 10 on the History exam.

In [59]:

```
tudents["Karl"] = {"Gender": "Male", "Algebra": 14, "History": 10}
tudents
```

Out[59]:

```
{'Amadeus': {'Algebra': 8, 'Gender': 'male', 'History': 13},
 'Karl': {'Algebra': 14, 'Gender': 'Male', 'History': 10},
 'Ludwig': {'Algebra': 12, 'Gender': 'Other', 'History': 18},
 'Mona': {'Algebra': 6, 'Gender': 'Female', 'History': 27},
 'Rosa': {'Algebra': 19, 'Gender': 'Female', 'History': 22}}
```

e) Use a `for` -loop to print out the names and scores of all students on the screen. The output should look like something this (the order of the students doesn't matter):

```
Student Amadeus scored 8 on the Algebra exam and 13 on the History exam
Student Rosa scored 19 on the Algebra exam and 22 on the History exam
...
```

[Hint: Dictionaries are iterables, also, check out the `items` function for dictionaries.]

In [60]:

```
or k, v in students.items():
    a = 'Student {0} scored {1} on the Algebra exam and {2} on the History exam'.format
k, v['Algebra'], v['History'])
    print(a)
pass
```

```
Student Amadeus scored 8 on the Algebra exam and 13 on the History exam
Student Rosa scored 19 on the Algebra exam and 22 on the History exam
Student Mona scored 6 on the Algebra exam and 27 on the History exam
Student Ludwig scored 12 on the Algebra exam and 18 on the History exam
Student Karl scored 14 on the Algebra exam and 10 on the History exam
```

f) Use a dict comprehension and the lists `names` and `short_long` from assignment 2 to create a dictionary of names and wether they are short or long. The result should be a dictionary equivalent to {'Forex':'long', 'Tesco':'long', ...}.

[Note: Remember that dictionaries in Python are unordered and that the order of the pairs in the above dictionary is arbitrary, you might not get the same order, this is fine.]

In [61]:

```
ictionary = {name: length for (name, length) in zip(names, short_long)}
ictionary
```

Out[61]:

```
{'Alonzo': 'long', 'Forex': 'long', 'Tesco': 'long', 'Zeno': 'short'}
```

5. Introductory file I/O

File I/O in Python is a bit more general than what most R programmers are used to. In R, reading and writing files are usually performed using file type specific functions such as `read.csv` while in Python we usually start with reading standard text files. However, there are lots of specialized functions for different file types in Python as well, especially when using the **pandas (http://pandas.pydata.org/)** library which is built around a datatype similar to R DataFrames. Pandas will not be covered in this course though.

[Literature: Files are introduced in LP part II chapter 4 and chapter 9.]

The file `students.tsv` contains tab separated values corresponding to the students in previous assigments.

a) Iterate over the file, line by line, and print each line. The result should be something like this:

```
Amadeus  Male    8   13
Rosa Female  19  22
...
```

The file should be closed when reading is complete.

[Hint: Files are iterable in Python.]

In [65]:

```
ata = open("students.tsv", "r")

or each in data:
   print(each)
ata.close()
```

```
madeus Male    8       13

osa    Female  19      22

ona    Female  6       27

udwig  Other   12      18

arl    Male    14      10
```

b) Working with many files can be problematic, especially when you forget to close files or errors interrupt programs before files are closed. Python thus has a special `with` statement which automatically closes files for you, even if an error occurs. Redo the assignment above using the `with` statement.

[Literature: With is introduced in LP part II chapter 9 page 294.]

In [66]:

```
ith open("students.tsv", "r") as data2:
   for each in data2:
       print(each)
```

```
madeus Male    8       13

osa    Female  19      22

ona    Female  6       27

udwig  Other   12      18

arl    Male    14      10
```

c) Recreate the dictionary from assignment the previous assignment by reading the data from the file. Using a dedicated csv-reader is not permitted.

In [67]:

```
ata3 = open("students.tsv", "r")
ictionary = {}

or each in data3:
    name, sex, algebra, history = each.split()
    dictionary[name] = {'Gender': sex, 'Algebra': algebra, 'History': history}

ictionary
```

Out[67]:

```
{'Amadeus': {'Algebra': '8', 'Gender': 'Male', 'History': '13'},
 'Karl': {'Algebra': '14', 'Gender': 'Male', 'History': '10'},
 'Ludwig': {'Algebra': '12', 'Gender': 'Other', 'History': '18'},
 'Mona': {'Algebra': '6', 'Gender': 'Female', 'History': '27'},
 'Rosa': {'Algebra': '19', 'Gender': 'Female', 'History': '22'}}
```

d) Using the dictionary above, write sentences from task 4e above to a new file, called `students.txt` .

In [68]:

```
extfile = open("students.txt", 'w')
or k, v in students.items():
    a = 'Student {0} scored {1} on the Algebra exam and {2} on the History exam'.format
k, v['Algebra'], v['History'])
    textfile.write(a + '\n')

extfile.close()
```