Overview

day one

0. introduction
1. text output and manipulation
2. reading and writing files

today

3. lists and loops
4. writing functions

day three

5. conditional statements
6. dictionaries

day four

7. files, programs and user input
8. biopython

day five

hands on training

feedback and discussion

Etherpad for today:
http://python-from-scratch.pad.spline.de/2

This course (apart from chapter 8) is based on the book "**Python for Biologists**":

http://pythonforbiologists.com/

python™ from scratch

A primer for scientists working with Next-Generation-Sequencing data

CHAPTER 3

lists and loops

Chapter 3: lists and loops

lists and loops help us deal with these issues:

- performing tasks repeatedly for multiple, similar objects
- usually not possible to know the number of data items
- necessary for dealing with large datasets

Dealing with repetitive tasks

often the **same task** needs to be performed **repeatedly** for a **set of similar objects** with only slight variation for each object

lists can hold a variable number of elements

loops enable running the same commands multiple times

Creating lists and retrieving elements

- creating lists:

```
apes = ["Homo","Pan","Gorilla"]
conserved_sites = [24,56,132]
```

- each list now has 3 elements
- to get an element from a list, use its index like this:

```
print(apes[0])
site = conserved_sites[2]
```

- to get the index of an element use the index method:

```
chimp_index = apes.index("Pan")
# chimp_index is now 1
```

Splitting a string into a list

If a string is structured by separators, we can use the **split** method to create a list from it:

```
>>> data_text = "comma,separated,values"
>>> data_list = data_text.split(",")
>>> data_list
["comma","separated","values"]
```

The separator is not important, though:

```
>>> sentence = "This is a sentence"
>>> sentence.split(" ")
["This","is","a","sentence"]
```

Use the split method when dealing with **CSV files**

Accessing list elements

- directly access the last element of a list by index -1:

```
last_ape = apes[-1]
```

- retrieve a sequence of elements by start and end index:

```
ranks = ["kingdom","phylum","class","order","family"]
lower_ranks = ranks[2:5]
# lower ranks are class, order and family
```

(start is inclusive, end is exclusive)

- remember this from strings?
that's right – **strings behave a lot like lists**!

Modifying lists

- add **one** element to a list with the append method:

```
apes = ["Homo","Pan","Gorilla"]
apes.append("Pongo")
```

- add **more than one** element with the extend method:

```
apes = ["Homo","Pan","Gorilla"]
apes.extend(["Pongo","Gibbon"])
```

- concatenate two lists:

```
apes = ["Homo","Pan","Gorilla"]
monkeys = ["Papio","Macaca"]
primates = apes + monkeys
```

Modifying lists (pt. 2)

- invert the order of a list with the reverse method:

```
ranks = ["kingdom","phylum","class","order","family"]
ranks.reverse()
# ranks are now in reverse order
```

- sort a list (alphabetically) with the sort method

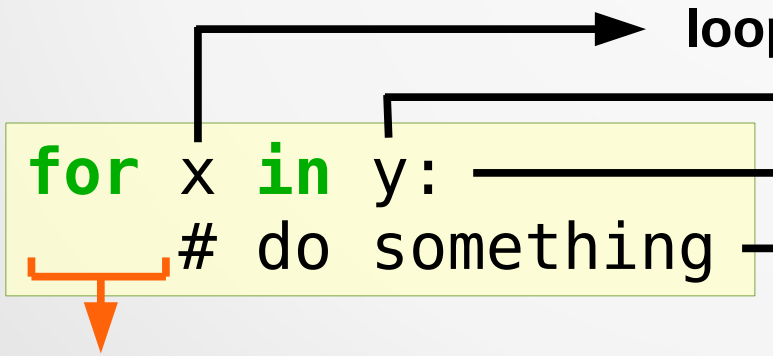
```
ranks.sort()
# new order: class, family, kingdom, ...
```

Introducing loops

- loop over elements in a list using a **for** loop:

```
apes = ["Homo","Pan","Gorilla"]
for ape in apes:
    print(ape + " is an ape")
```

- physiology of a **for** loop:



loop variable: changes with every iteration

list of elements: does not change

loop header (always ended by a **colon**)

loop body: here the loop variable x can be used (with a different value in each iteration)

mind the gap!
the *indentation* is used to distinguish the **loop body** from the **loop header**

blocks and scopes

- In python (and most other programming languages) a variables live in a context, called a scope
- scopes are also called blocks of code
- code blocks are denoted by indentations (spaces or tabs)
- indentations are flexible, but every line in a scope must have exactly the same indentation!
- examples for scopes are
  - the global scope, actually called module scope
  - functions and methods
  - loops

indentation matters – a lot!

- using different kinds of indentation (a mix of spaces and TABs) **will** result in `IndentationErrors`
  → **never mix spaces and TABs**
- it is best practice to use four spaces for indentation
- make sure that you configure your editor to use consistent indentation (often called "tab emulation")
- take care when exchanging scripts with other people

Looping over a string

- strings behave like lists in a lot of ways.
- we can also loop over the characters of a string:

```
for letter in "ACGT":
    print("one letter is " + letter)
```

iterating over lines in a file

- a file can be read line by line using a **for** loop

```
file = open("some_input.txt")
for line in file:
    print(line)
```

- advantageous when dealing with *structured* files or *very large* files (only one line is read into memory in each iteration)

looping with ranges

- run some commands a certain number of times

```
for number in range(6):
    print(number)
```

```
protein = "vlspadktnv"
for end_pos in range(3,8):
    print(protein[0:end_pos])
```

- note: end number given to `range` is exclusive

Recap

In this unit you have learned

- how to loop over a list of elements using a for loop
- the importance of indentations to denote blocks of code
- the notion of a loop variable and variable scopes
- reading files line by line
- iterating over number ranges

splitting a string into a list

- essential when working with structured files (like CSV)
- splitting a string is done with the `split` function

```
names = "mus,rattus,cricetus,cavia"
rodents = names.split(",")
print(str(rodents))
```

- a delimiting character or string must be specified (here: `","`)

Exercise 3-1: Processing DNA in a file

- file `ex3-1_input.txt` contains a number of DNA sequences
- the **first 14 positions** are identical and represent a **sequencing adapter** which is supposed to be removed

  a) **trim** the adapter sequence off, write the trimmed sequences to a (single) new file

  b) print out the **length** of each sequence to the screen

Exercise 3-1: Processing DNA in a file

expected output:
If your scripts works as intended you should get printed **output in the terminal** like this:

```
processed sequence with length 43
processed sequence with length 38
processed sequence with length 49
processed sequence with length 34
processed sequence with length 48
```

Additionally a **file** should have been **created** like this one:

trimmed.txt

```
TCGATCGATCGATCGATCGATCGATCGATCGATCGATC
ACTGATCGATCGATCGATCGATCGATGCTATCGTCGT
ATCGATCACGATCTATCGTACGTATGCATATCGATATCGATCGTAGTC
ACTATCGATGATCTAGCTACGATCGTAGCTGTA
ACTAGCTAGTCTCGATGCATGATCAGCTTAGCTGATGATGCTATGCA
```

Exercise 3-2: Exons from genomic DNA

- file `genomic_dna.txt` contains a section of genomic DNA
- file `exons.txt` contains a list of start/stop positions of exons (one line each, start/stop separated by a comma)
- Write a program that will
  - **extract** the exon segments
  - **concatenate** them, and
  - **write** them to a new file.

Exercise 3-2: Exons from genomic DNA

expected output:
If your scripts works as intended a **file** should have been **created** like this one:

coding_sequence.txt

```
CGTACCGTCGACGATGCTACGATCGTCGATCGTAGTCGATCATCGATCG
ATCGCGATCGATCGATATCGATCGATATCATCGATGCATCGATCATCGA
TCGATCGATCGATCGACGATCGATCGATCGTAGCTAGCTAGCTAGATCG
ATCATCATCGTAGCTAGCTCGACTAGCTACGTACGATCGATGCATCGAT
CGTACGATCGATCGATCGATCGATCGATCGATCGATCGATCGATCGTAG
CTAGCTACGATCG
```

Getting help

- Python interactive help