

Introduction to



– with Application to Bioinformatics

**NBS**

```
Scratch — (179 x 95)
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from etils import time_me, print_args, print_retval
from etils import get_gtf_value
from etils.rna import RNATranslationTable
import re
#from os.path import splitext

import logging
logger = logging.getLogger() # root logger
logging.basicConfig(level=logging.INFO, format="%{message}s")

@time_me
@print_args
def get_all_transcripts(filename="Homo_sapiens.GRCh38.B7.gtf", chromosome="7", gene="PNUCC0000001631"):

    transcripts = {}

    # First pass: Fetch all transcripts for the given gene and chromosome
    # -----
    logger.debug('First pass on file %s' % filename)
    logger.debug('Chr: %s | Gene %s' % (chromosome, gene))
    with open(filename, mode="rt") as gtf:
        #gene_id = 'gene_id "%s"' % gene
        gene_re = re.compile(r'gene_id"%s"?{3}?' .format(gene))
        for line in gtf:
            blocks = line.split("\t")
            # Only that chromosome and
            if (
                len(blocks) < 3 or          # no comments, please
                blocks[0] != chromosome or # only that chromosome. Careful: not comparing integers!
                blocks[2] != "transcript" or # the line should be a transcript
                not gene_re.search(blocks[8]) # Is that the right gene?
            ):
                continue # skip to the next line

            # Otherwise, it is a transcript for the given gene and chromosome
            attributes = blocks[8]
            transcript_id = get_gtf_value("transcript_id", attributes)

            assert( transcript_id ) # is not None
            assert( transcript_id not in transcripts), ("How come I see transcript %s already? \n\nline:\n\n%s" % (transcript_id, line))

            start = int(blocks[3])
            end = int(blocks[4])
            strand = 1 if blocks[6] == '+' else -1

            # Adding it to the table
            transcripts[transcript_id] = {
                'start': start,
                'end': end,
                'strand': strand,
                'exons': {}, # exons will be added in the second pass. Empty so far.
                'start_codon': None,
                'stop_codon': None
            }
            logger.debug('Added records: {} - {}'.format(transcript_id, transcripts[transcript_id]))

    logger.debug('Transcripts after first pass')
    logger.debug(transcripts)

    # Second pass, fetching the exons for these transcripts
    # Must rescan, can't reuse the gtf iterator: it's at the end already.
    # -----
    logger.debug('Second pass')
    with open(filename, mode="rt") as gtf:
        for line in gtf:
            blocks = line.split("\t")

            if (
                len(blocks) < 3 or          # no comments, please
                blocks[0] != chromosome or # only that chromosome
                not (blocks[2] == "exon" or blocks[2] == "start_codon" or blocks[2] == "stop_codon")
            ):
                continue # Skip that line

            feature = blocks[2]
            attributes = blocks[8]

            transcript_id = get_gtf_value("transcript_id", attributes)

            if transcript_id not in transcripts: # checking the keys
                continue # Skip cuz not a transcript for that given gene

            if not gene_re.search(attributes):
                print("Weird! I should have a gene_id (%gene) in attr:".format(gene=gene, attr=attributes))

            if feature == "exon":
                logger.debug('Found an exon')
                exon_id = get_gtf_value("exon_id", attributes)
                exons = transcripts[transcript_id].get('exons', None)
                assert( exons is not None )
                if exon_id in exons:
                    print("Weird! Have I seen that exon %s before?" % exon_id)


```



```

@time_me
@print_args
def get_all_transcripts(filename="Homo_sapiens.GRCh38.87.gtf", chromosome='7', gene='ENSG0000001626'):
    transcripts = {}

    # First pass: Fetch all transcripts for the given gene and chromosome
    # =====
    logger.debug('First pass on file %s' % filename)
    logger.debug('Chr %s | Gene %s' % (chromosome, gene))
    with open(filename, mode="rt") as gtf:
        # gene_id = 'gene_id "%s"' % gene
        gene_re = re.compile(r'gene_id\s+"?{"?"?.format(gene))
        for line in gtf:
            blocks = line.split("\t")
            # Only that chromosome and
            if (
                len(blocks) < 9 or
                blocks[0] != chromosome or
                blocks[2] != 'transcript' or
                not gene_re.search(blocks[8])
            ):
                # no comments, please
                # only that chromosome. Careful: not comparing integers!
                # the line should be a transcript
                # Is that the right gene?
                continue # skip to the next line

            # Otherwise, it is a transcript for the given gene and chromosome
            attributes = blocks[8]
            transcript_id = get_gtf_value('transcript_id', attributes)

            assert( transcript_id ) # is not None
            assert( transcript_id not in transcripts), ("How come I see transcript %s already" % (transcript_id))

            start = int(blocks[3])
            end = int(blocks[4])
            strand = 1 if blocks[6] == '+' else -1

            # Adding it to the table
            transcripts[transcript_id] = {
                'start': start,
                'end': end,
                'strand': strand,
                'exons': {}, # exons will be added in the second pass. Empty so far.
                'start_codon': None,
                'stop_codon': None
            }
            logger.debug('Added record: {} => {}'.format(transcript_id, transcripts[transcript_id]))

    logger.debug('Transcripts after first pass')
    logger.debug(transcripts)

    # Second pass: fetching the exons for these transcripts

```

Variables

Functions

Arguments

Control Flow

I/O

RegExp

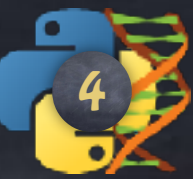
...

# Why Python?

Repetitive task: automate!

- \* Write code
- \* transform it to executable ( compile )
- \* run
- \* check/test
- \* start again...

Too slow... => Python is for you



# Why Python?

Repetitive task: automate!

- \* Write code
- \* transform it to executable ( compile )
- \* run
- \* check/test
- \* start again...

Too slow... => Python is for you

Shell? Sure!

But mostly for moving files around  
and updating text data  
=> not for every task.

Want to

- \* get some data
- \* store them temporarily  
to manipulate in some sort of map

...shells are limited.



# Why Python?

Repetitive task: automate!

- \* Write code
- \* transform it to executable ( compile )
- \* run
- \* check/test
- \* start again...

Too slow... => Python is for you

Shell? Sure!

But mostly for moving files around  
and updating text data  
=> not for every task.

Want to

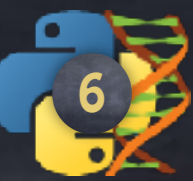
- \* get some data
- \* store them temporarily  
to manipulate in some sort of map

...shells are limited.

Python has built-in constructs.

Python offers more structure,  
but allowing splitting programs into modules

Python is multi-purpose.



# Release dates (Wikipedia)

- Python 1.0 - January 1994
  - Python 1.5 - December 31, 1997
  - Python 1.6 - September 5, 2000
- Python 2.0 - October 16, 2000
  - Python 2.1 - April 17, 2001
  - Python 2.2 - December 21, 2001
  - Python 2.3 - July 29, 2003
  - Python 2.4 - November 30, 2004
  - Python 2.5 - September 19, 2006
  - Python 2.6 - October 1, 2008
  - Python 2.7 - July 3, 2010
- Python 3.0 - December 3, 2008
  - Python 3.1 - June 27, 2009
  - Python 3.2 - February 20, 2011
  - Python 3.3 - September 29, 2012
  - Python 3.4 - March 16, 2014
  - Python 3.5 - September 13, 2015
  - Python 3.6 - December 23, 2016

dead

Incompatibilities





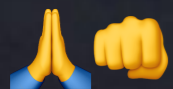
Fred



Fred



Thomas



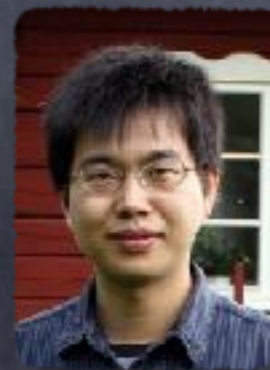
Johan



Åsa



Moritz



Nanjiang

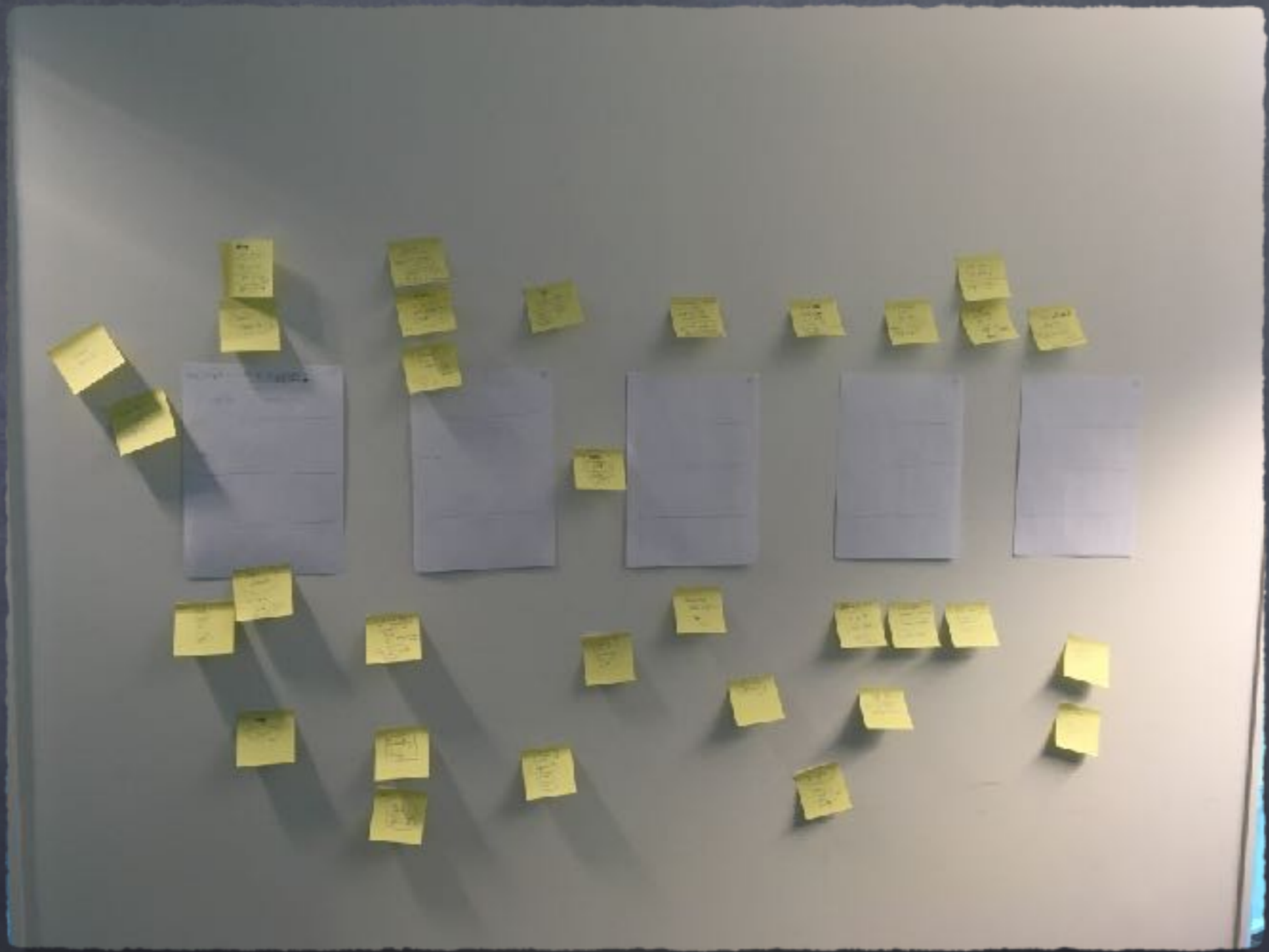


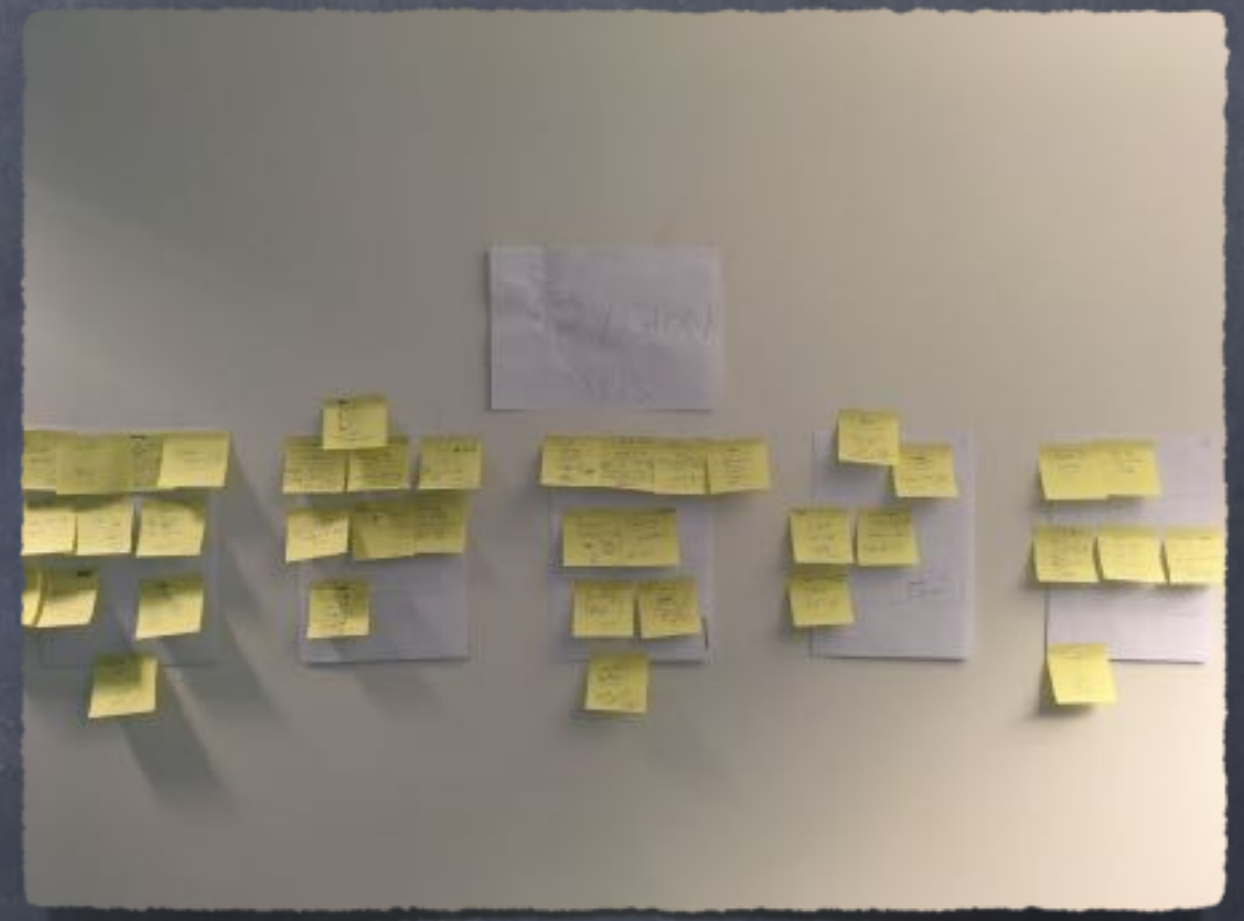
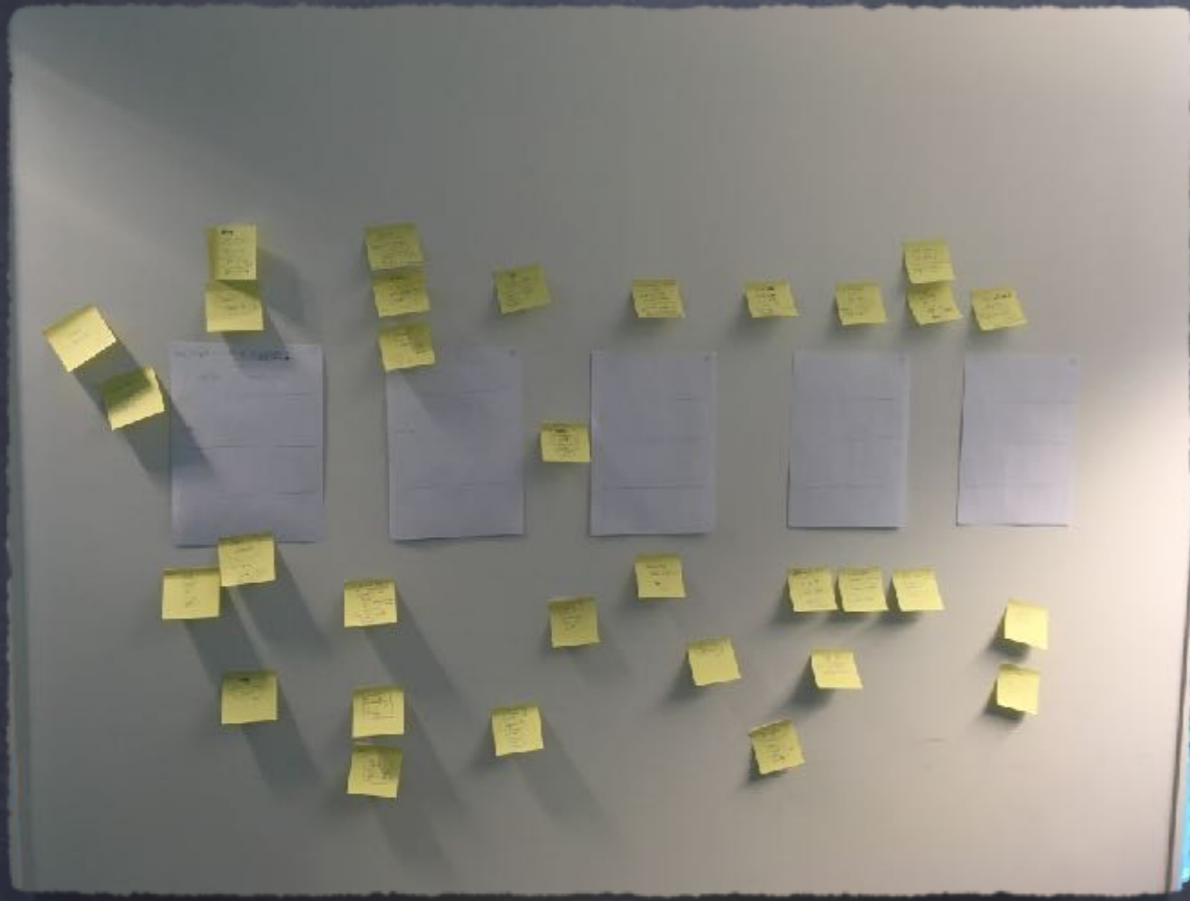
Anders



Markus







	Monday	Tuesday	Wednesday	Thursday	Friday
09:00-12:00	Lectures				
12:00-13:00	Lunch				
13:00-17:00	Exercises + Project			Assistants	

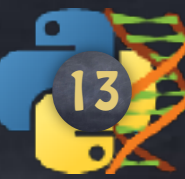
Eh... 5 days only?

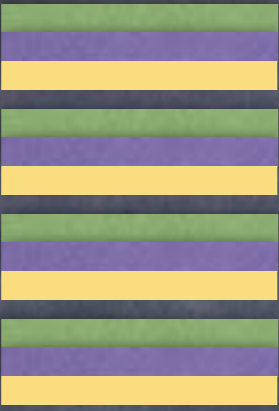

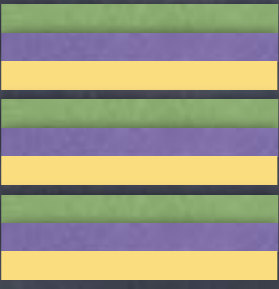


Adapted pedagogy

1. Explain something
2. Example
3. Practice → in pairs

Repeat

Ideally, the assistants sleep



	Monday	Tuesday	Wednesday	Thursday	Friday
09:00-12:00		<u>Hands-on Python stuff</u>			
12:00-13:00	Lunch				
13:00-15:00		<u>Hands-on more Python stuff</u>			
15:00-17:00	Own Practice on Main Assignment				

Topic

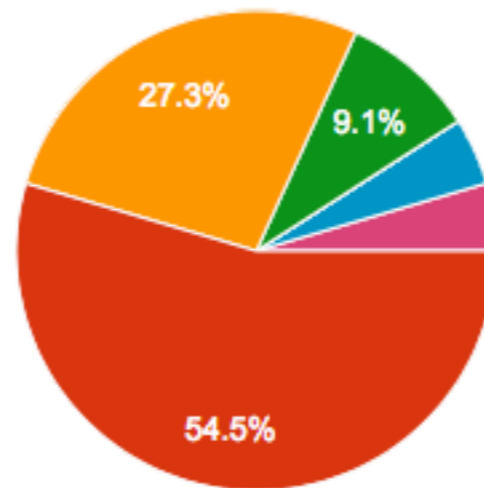
Example (me)

Practice (You!)

# Who you are...

## Position

22 responses

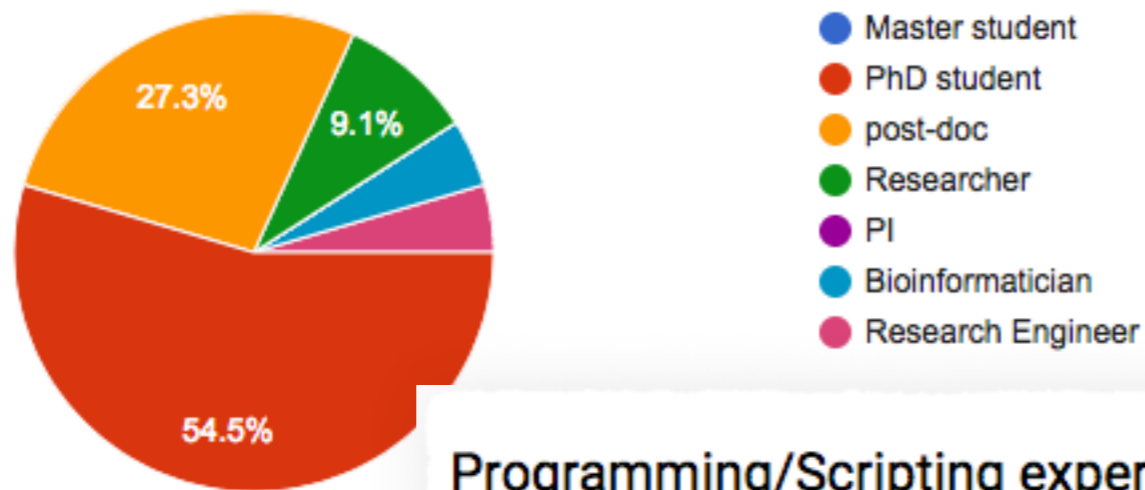


- Master student
- PhD student
- post-doc
- Researcher
- PI
- Bioinformatician
- Research Engineer

# Who you are...

## Position

22 responses



## Programming/Scripting experience

22 responses

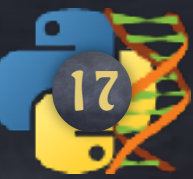




Work in pairs

Raise your hand for help

Ask politely



Introduction to Python | HT17

Secure | <https://nbsweden.github.io/PythonCourse/ht17/>

NBS

Topics Project Preliminaries Help

## Introduction to Python - HT17

This course provides a practical introduction to the writing of Python programs for the complete novice. Participants are lead through the core aspects of Python illustrated by a series of example programs. Upon completion of the course, attentive participants will be able to write simple Python programs from scratch and to customize more complex code to fit their needs.

The craft of programming is about taking design decisions to avoid overwhelming complexity and permit easy maintenance over time, insuring reliability (which goes beyond debugging) and utilizing computer resources efficiently.

- The course is suitable for complete beginners and assumes no prior programming experience (beyond the ability to use a text editor).
- A very basic knowledge of LINUX would be an advantage, such as navigating through folders and issuing commands at a shell prompt. We will not teach Linux in detail. Other courses are available at SciLifeLab for 2.

Before the first lecture, we require you to follow these [preliminary steps](#).

### » Schedule

From Monday October 9<sup>th</sup>, to Friday October 13<sup>th</sup> 2017 (week 41).

- 09:00 - 12:00: Lectures + Hands-on (including a Fika break)
- 12:00 - 13:00: Lunch
- 13:00 - 15:00: Lectures + Hands-on
- 15:00 - 17:00: Practice time (with assistants)

### » Course Content

During this course, you will learn about:

- Core concepts about Python syntax: Data types, blocks and indentation, variable scoping, iteration, functions, methods and arguments
- Different ways to control program flow using loops and conditional tests
- Regular expressions and pattern matching
- Writing functions and best-practice ways of making them usable
- Reading from and writing to files
- Code packaging and Python libraries
- How to work with biological data using external libraries (if time allows).

### » Learning Outcomes

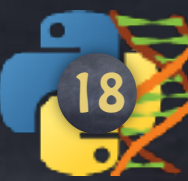
After this course you should be able to:

- Edit and run Python code
- Write file-processing python programs that produce output to the terminal and/or external files.
- Create stand-alone python programs to process biological data
- Know how to develop your skills in Python after the course (including debugging)

**Learning objectives (ie goals for the teachers)**

- Increase the student's toolbelt for better quality and performance at work
- Make students understand that there is more to programming than only knowing the syntax of a language. This expertise is precisely what NBS provides.

### » Format





Topics



Project



Preliminaries



Help

## Introduction to Python - HT17

This course provides a practical introduction to the writing of [Python](#) programs for the complete novice. Participants are lead through the core aspects of Python illustrated by a series of example programs. Upon completion of the course, attentive participants will be able to write simple Python programs from scratch and to customize more complex code to fit their needs.

The craft of programming is about taking design decisions to avoid overwhelming complexity and permit easy maintenance over time, insuring reliability (which goes beyond debugging) and utilizing computer resources efficiently.

- The course is suitable for complete beginners and assumes no prior programming experience (beyond the ability to use a text editor).
- A very basic knowledge of UNIX would be an advantage, such as navigating through folders and issuing commands at a shell prompt. We will not teach Unix in detail: Other course are available at SciLifeLab for it.

Before the first lecture, we require you to follow these [preliminary steps](#).

Introduction to Python - HT17

**NBS** Topics Project Preliminaries Help

## About your main assignment

**Background:** For many diseases with known causative mutations, screening methods have been developed to detect whether people have a high risk of becoming sick, even before the onset of the actual disease.

Over the last few years, the cost of full genome sequencing has gone down so that, in some cases, it might be cheaper to collect the complete genome sequence of patients with a high risk of carrying variants associated with the disease, rather than using targeted screening procedures.

Cystic fibrosis is a complex disease, where patients often manifest the following symptoms: problems with lung functions, diabetes and infertility. From a genetic point of view, there are several mutations associated with this disease. In particular, the CFTR gene (short for Cystic Fibrosis Transmembrane Conductance Regulator) encodes an ion channel protein acting in epithelial cells, and carries several non-synonymous genetic variants, with alterations leading to premature stop codons, that are known to cause the disease.

**Goal:** In this assignment, you have access to the human reference genome as well as the genome annotation. In addition, you have full genome sequence data from five individuals from a family at risk of carrying mutations related to the disease.

Your task is to write a Python program that will extract the CFTR gene, translate the gene sequence to its corresponding amino-acid sequence and based on the inferred amino-acid sequence determine whether any of the five given individuals is affected.

### » Fetch the appropriate files

The main task is divided in several steps. The first step is to fetch the sequence file (in `fasta` format) and the appropriate annotation file (in `gtf` format) from the [Ensembl database](#).

The CFTR gene is chromosome 7.

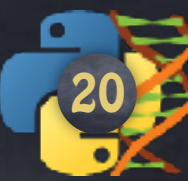
### » Warmup

1. What is the length of the chosen DNA sequence?  
▶ Tip
2. How many genes are annotated in the GTF file?  
▶ Note
3. What fraction of the chromosome is annotated as genes?

### » Architect a method

All the following tasks are now related to the CFTR gene.

In the annotation file (from the Ensembl database), that gene has the id `ENSC0000001626` on chromosome 7.



## » Course Content

During this course, you will learn about:

- Core concepts about Python syntax: Data types, blocks and indentation, variable scoping, iteration, functions, methods and arguments
- Different ways to control program flow using loops and conditional tests
- Regular expressions and pattern matching
- Writing functions and best-practice ways of making them usable
- Reading from and writing to files
- Code packaging and Python libraries
- How to work with biological data using external libraries (if time allows).

## » Learning Outcomes

After this course you should be able to:

- Edit and run Python code
- Write file-processing python programs that produce output to the terminal and/or external files.
- Create stand-alone python programs to process biological data
- Know how to develop your skills in Python after the course (including debugging)

### Learning objectives (ie goals for the teachers)

- Increase the student's toolbelt for better quality and performance at work
- Make students understand that there is more to programming than only *knowing* the syntax of a language. This expertise is precisely what **NBIS** provides.

Syntax

||

Programming

Stream of characters

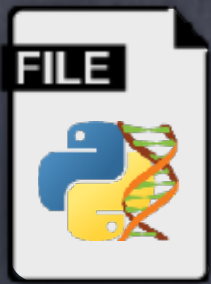


Stream of tokens



Abstract  
Syntax  
Tree





One Letter  
at a time

Lexer

illegal word

Words

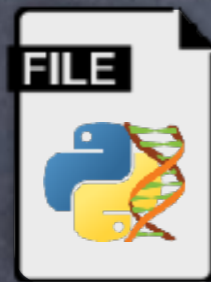
Parser

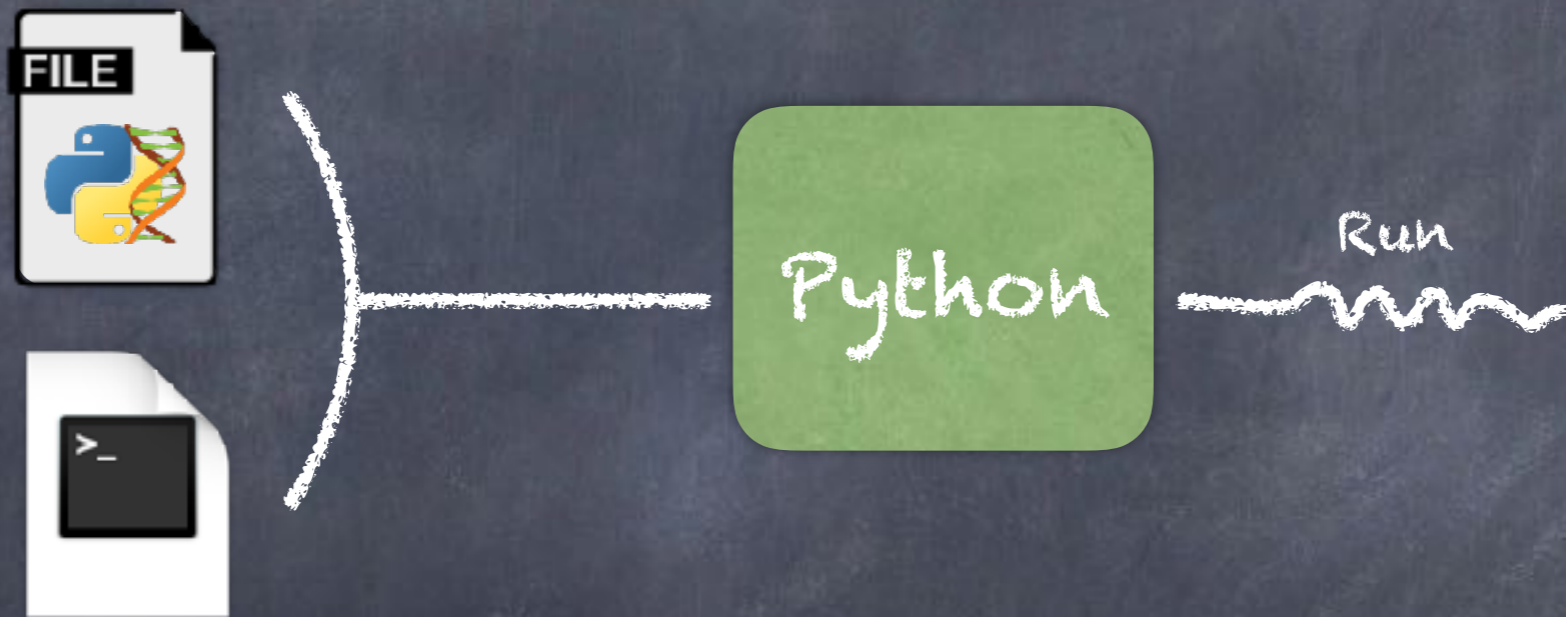
Proper  
syntax?

Analys

Run







Example: 1<sup>st</sup> jupyter notebook

# So far...

---

## builtin types

int

float

str

List

## Operations

+, -, \*, /, \*\*, %, // ...

+, -, \*, /, ...

word[3], word[2:5]

List[2:-3], List[2:3]=['a','b']



# So far...

---

## builtin types

int

float

str

List

## Operations

+, -, \*, /, \*\*, %, // ...

+, -, \*, /, ...

word[3], word[2:5]

List[2:-3], List[2:3]=['a','b']

# Explicit line joining

"something " \

'over ' \

"several " \

'lines.'

# Implicit line joining

List = ["something ",

'over ',

"several ",

'lines.']

# Comments

---

```
# A traditional one line comment
```

```
"""
```

```
Any string not assigned to a variable is  
considered a comment.
```

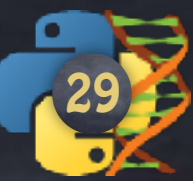
```
This is an example of a multi-line comment.
```

```
"""
```

```
"This is a single line comment"
```

```
# Blank lines are ignored
```

```
# and the beginning of a line matters
```



# Literals

---

Values like

'Hello' "hi"

3

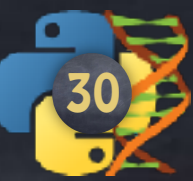
3.14

"3"

'3.14'

'file.txt'

they have a type



# Identifiers

---

for example: variables  
functions  
modules  
classes

# Identifiers

for example: variables  
functions  
modules  
classes

Identifiers are unlimited in length. Case is significant.

```
identifier ::= xid_start xid_continue*  
id_start  ::= <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and characters with the Other_ID_Start property>  
id_continue ::= <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc and others with the Other_ID_Continue property>  
xid_start  ::= <all characters in id_start whose NFKC normalization is in "id_start xid_continue*">  
xid_continue ::= <all characters in id_continue whose NFKC normalization is in "id_continue*">
```

The Unicode category codes mentioned above stand for:

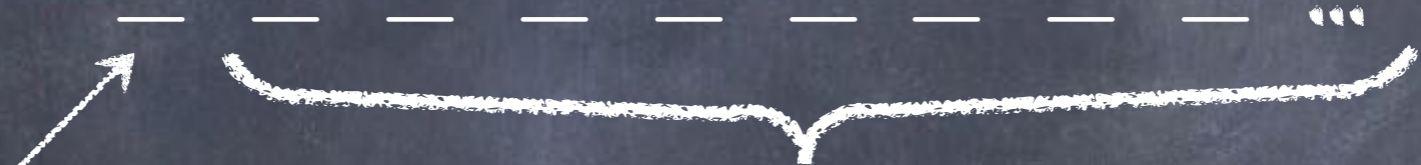
- *Lu* – uppercase letters
- *Ll* – lowercase letters
- *Lt* – titlecase letters
- *Lm* – modifier letters
- *Lo* – other letters
- *Nl* – letter numbers
- *Mn* – nonspacing marks
- *Mc* – spacing combining marks
- *Nd* – decimal numbers
- *Pc* – connector punctuations
- *Other\_ID\_Start* – explicit list of characters in [PropList.txt](#) to support backwards compatibility
- *Other\_ID\_Continue* – likewise



# Identifiers

---

for example: variables

- 
- ✓ a letter
  - ✓ underscore
  - x digit
  - ✓ all letters (uppercase, lowercase)
  - ✓ all digits
  - ✓ the underscore

# Identifiers

for example: variables

- 
- ✓ a letter
  - ✓ underscore
  - x digit
  - ✓ all letters (uppercase, lowercase)
  - ✓ all digits
  - ✓ the underscore

No

+ - \* \$ % ; : , ? ! { } ( ) < > " ' | \ @

etc...

# Keywords

---

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

# Keywords

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

Identifiers of the form:

- `__*` # special meaning for modules
- `__*__` # System-defined names
- `__*` # special meaning for classes

# Online definitions

strings ⇨

integers ⇨

floats ⇨

operators ⇨

delimiters ⇨

# Standard Library

		Built-in Functions		
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

Example with `str`

# range ()

## 4.6.6. Ranges

The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

`class range(stop)`

`class range(start, stop[, step])`

The arguments to the `range` constructor must be integers (either built-in `int` or any object that implements the `__index__` special method). If the `step` argument is omitted, it defaults to 1. If the `start` argument is omitted, it defaults to 0. If `step` is zero, `ValueError` is raised.

For a positive `step`, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative `step`, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than `sys.maxsize` are permitted but some features (such as `len()`) may raise `OverflowError`.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

**start**

The value of the `start` parameter (or 0 if the parameter was not supplied)

**stop**

→ Notebook 2

# Iteration - for Loop

Iterable

```
for x in range(10): #0-9
    # Do something on x
    print('Item:', x)
```

Indentation with Tab character

```
fruits = ['Apple', 'Orange']

for fruit in fruits:
    print(fruit)
```



# while Loop

---

```
x = 0
while x < 100:
    print(x)
x += 1 ?
```

Operations on

- \* numbers
- \* strings
- \* Lists
- ...

```
a = <choose>
```

```
b = <choose>
```

```
print(a <op> b)
```

Operation	Result
<code>x + y</code>	sum of $x$ and $y$
<code>x - y</code>	difference of $x$ and $y$
<code>x * y</code>	product of $x$ and $y$
<code>x / y</code>	quotient of $x$ and $y$
<code>x // y</code>	floored quotient of $x$ and $y$
<code>x % y</code>	remainder of $x / y$
<code>-x</code>	$x$ negated
<code>+x</code>	$x$ unchanged
<code>abs(x)</code>	absolute value or magnitude of $x$
<code>int(x)</code>	$x$ converted to integer
<code>float(x)</code>	$x$ converted to floating point
<code>complex(re, im)</code>	a complex number with real part $re$ , imaginary part $im$ . $im$ defaults to zero.
<code>c.conjugate()</code>	conjugate of the complex number $c$
<code>divmod(x, y)</code>	the pair <code>(x // y, x % y)</code>
<code>pow(x, y)</code>	$x$ to the power $y$
<code>x ** y</code>	$x$ to the power $y$

Operation	Result
<code>math.trunc(x)</code>	$x$ truncated to <i>Integral</i>
<code>round(x[, n])</code>	$x$ rounded to $n$ digits, rounding half to even. If $n$ is omitted, it defaults to 0.
<code>math.floor(x)</code>	the greatest <i>Integral</i> $\leq x$
<code>math.ceil(x)</code>	the least <i>Integral</i> $\geq x$

# Comparators

Logical  
identity

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Physical  
identity

int, float, str

# On sequences

eg strings or lists

Operation	Result
<code>x in s</code>	True if an item of <i>s</i> is equal to <i>x</i> , else False
<code>x not in s</code>	False if an item of <i>s</i> is equal to <i>x</i> , else True
<code>s + t</code>	the concatenation of <i>s</i> and <i>t</i>
<code>s * n</code> or <code>n * s</code>	equivalent to adding <i>s</i> to itself <i>n</i> times
<code>s[i]</code>	<i>i</i> th item of <i>s</i> , origin 0
<code>s[i:j]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i>
<code>s[i:j:k]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> with step <i>k</i>
<code>len(s)</code>	length of <i>s</i>
<code>min(s)</code>	smallest item of <i>s</i>
<code>max(s)</code>	largest item of <i>s</i>
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <i>x</i> in <i>s</i> (at or after index <i>i</i> and before index <i>j</i> )
<code>s.count(x)</code>	total number of occurrences of <i>x</i> in <i>s</i>

# On <sup>mutable</sup> sequences

eg ~~strings~~ or lists

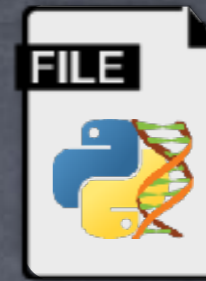
Operation	Result
<code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list
<code>s.append(x)</code>	appends <i>x</i> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code> )
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code> )
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code> )
<code>s.extend(t)</code> or <code>s += t</code>	extends <i>s</i> with the contents of <i>t</i> (for the most part the same as <code>s[len(s):len(s)] = t</code> )
<code>s *= n</code>	updates <i>s</i> with its contents repeated <i>n</i> times
<code>s.insert(i, x)</code>	inserts <i>x</i> into <i>s</i> at the index given by <i>i</i> (same as <code>s[i:i] = [x]</code> )
<code>s.pop([i])</code>	retrieves the item at <i>i</i> and also removes it from <i>s</i>
<code>s.remove(x)</code>	remove the first item from <i>s</i> where <code>s[i] == x</code>
<code>s.reverse()</code>	reverses the items of <i>s</i> in place

# Special characters in Strings

Escape Sequence	Meaning	Notes
<code>\newline</code>	Backslash and newline ignored	
<code>\\</code>	Backslash (\)	
<code>\'</code>	Single quote (')	
<code>\"</code>	Double quote (")	
<code>\a</code>	ASCII Bell (BEL)	
<code>\b</code>	ASCII Backspace (BS)	
<code>\f</code>	ASCII Formfeed (FF)	
<code>\n</code>	ASCII Linefeed (LF)	
<code>\r</code>	ASCII Carriage Return (CR)	
<code>\t</code>	ASCII Horizontal Tab (TAB)	
<code>\v</code>	ASCII Vertical Tab (VT)	
<code>\ooo</code>	Character with octal value <i>ooo</i>	(1,3)
<code>\xhh</code>	Character with hex value <i>hh</i>	(2,3)



# Our first Python



Open a text editor:

- \* First line: `#!/usr/bin/python`
- \* Second line (optional): `# -*- coding: <some encoding> -*-`

Use a variable to store the following string items:

- \* Get the kids from school
- \* Buy groceries
- \* Fill up the car tank
- \* call mum
- \* Pay the electricity bill
- \* Read a Swedish book with å,ä,ö
- \* escape the special characters like \n and \t
- \* Call mum again

Open a text editor:

- \* First line: `#!/usr/bin/python`
- \* Second line (optional): `# -*- coding: <some encoding> -*-`

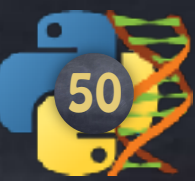
Use a variable to store the following string items:

- \* Get the kids from school
- \* Buy groceries
- \* Fill up the car tank
- \* call mum
- \* Pay the electricity bill
- \* Read a Swedish book with å,ä,ö
- \* escape the special characters like `\n` and `\t`
- \* Call mum again

Print a long line of 68 '=' symbols

For each item, print 'Remember to', a space, and then the item

Print again the same long line as above



# Our first Python file

```
first-loop.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

remember_todo = ['Get the kids from school',
                 'Buy groceries',
                 'Fill up the car tank',
                 'Call mum',
                 'Pay the electricity bill',
                 'Read a swedish book with å,ä,ö',
                 r'escape the special character like \n and \t',
                 'Call mum again', # Look...an extra comma...
                                   # No problem for Python
                ]

fake_line = '=' * 20 68

print( fake_line )

for item in remember_todo:
    print( 'Remember to', item.lower() )

print( fake_line )

U:--- first-loop.py All L25 (Python +3 MM) 17:40 1.32
(No changes need to be saved)
```

Operations on

- \* numbers
- \* strings
- \* lists
- \* IO files
- ...

```
open('filename', 'r', encoding='utf-8')
```

```
open('filename', 'r', encoding='utf-8')
```

file path

./some/folder/to/file/name  
some/folder/to/file/name  
/absolute/path/to/file/name  
../parent/search/to/file/name  
../../../../bla/bla/file/name

mode

'r' for read  
'w' for write  
'a' for append

Eh...guess...

Opening ...ok...  
How about closing ?

# Automatic closing

```
with open('filename', 'r', encoding='utf-8') as the_file:
    for line in the_file:      # the_file is iterable, yeii !
        print(line.rstrip())  # Removing the trailing \n
                               # since print() adds one.
```

## Control Flow

for and while loops

Conditional: if / else

# Conditional: if / else

Anything that evaluates to  
boolean of type bool  
True False

```
if condition:  
    print('This will be executed')  
else:  
    print('Otherwise, it is this one')
```

Indentation

else is optional

Evaluates to False:

False, None, 0, 0.0, []

... some more (later)

Otherwise True!



## Conditional: if / else

```
s = input('Want candies?\n')
if s == 'yes':
    print('Later!')
else:
    print('Go clean your room, anyway!')
```

## Conditional: if / else

```
s = input('Want candies?\n')
if s == 'yes':
    print('Later!')
else:
    print('Go clean your room, anyway!')
```

```
s = input('Want candies?\n')
if s in ['y', 'yes', 'Y', 'Yes', 'YES']:
    print('Later!')
else:
    print('Go clean your room, anyway!')
```

## Conditional: if / else

```
shopping = ['milk', 'eggs', 'bread', 'butter']
if len(shopping) > 2:
    print("I'll do it tomorrow")
else:
    print('ok, maybe today!')
```

## Conditional: if / else

```
shopping = ['milk', 'eggs', 'bread', 'butter']  
if len(shopping) > 2:  
    print("I'll do it tomorrow")  
else:  
    print('ok, maybe today!')
```

```
shopping = ['milk', 'eggs', 'bread', 'butter']  
if shopping:  
    print("Get on it")  
else:  
    print('Finito')
```

## Conditional: if / else

---

```
if foo:  
    if bar:  
        print(baz)  
    else:  
        print(q)
```

## Conditional: if / else

```
if foo:  
    if bar:  
        print(baz)  
    else:  
        print(q)
```

```
if foo:  
    if bar:  
        print(baz)  
else:  
    print(q)
```

# Nesting and Indentation

Read `book_chapter.txt` into Python, line by line,  
Print the first 50 characters of each line, appending an  
ellipsis ... at the end, if necessary.

# Nesting and Indentation

Read `book_chapter.txt` into Python, line by line,  
Print the first 50 characters of each line, appending an  
ellipsis ... at the end, if necessary.

```
with open('book_chapter.txt', 'r', encoding='utf-8') as file:
    for line in file:
        line = line.strip()
        if line:
            if len(line) > 50:
                print( line[0:50], '...' )
            else:
                print( line[0:50] ) # slicing gracefully,
                                    # but no ellipsis
```





**Goal:** In this assignment, you have access to the human reference genome as well as the genome annotation. In addition, you have full genome sequence data from five individuals from a family at risk of carrying mutations related to the disease.

Your task is to write a Python program that will extract the CFTR gene, translate the gene sequence to its corresponding amino-acid sequence and based on the inferred amino-acid sequence determine whether any of the five given individuals is affected.

## » Fetch the appropriate files

The main task is divided in several steps. The first step is to fetch the sequence file (in `fasta` format) and the appropriate annotation file (in `GTF` format) from the [Ensembl database](#).

The CFTR gene is chromosome 7.

## » Warmup

1. What is the length of the chosen DNA sequence?
  - ▶ Tip
2. How many genes are annotated in the GTF file?
  - ▶ Note
3. What fraction of the chromosome is annotated as genes?

## » Architect a method

All the following tasks are now related to the CFTR gene.

In the annotation file (from the [Ensembl database](#)), that gene has the id `ENSG0000001625` on chromosome 7.

1. How many transcripts can this gene generate?