

Lecture 3

Files/ Functions&Modules

Lecturer: Pieter De Bleser
Bioinformatics Core Facility, IRC



Files

Accessing file contents

- Two step process:
 - First we open the file
 - Then we access its contents
 - Read
 - Write
- When we are done, we close the file.

What happens at open() time?

- The system verifies
 - That you are an ***authorized user***
 - That you have the ***right permission***
 - ***Read permission***
 - ***Write permission***
 - Execute permission exists but doesn't apply
- and returns a ***file handle /file descriptor***

The file handle

- Gives the user
 - Direct access to the file
 - No directory look-ups
 - Authority to execute the file operations whose permissions have been requested

Python open()

- **open(name, mode = 'r', buffering = -1)**

where

- **name** is name of file
- **mode** is *permission requested*
 - Default is '*r*' for read only
- **buffering** specifies the *buffer size*
- *Use system default value (code -1)*

The modes

- Can request
 - **'r'** for read-only
 - **'w'** for write-only
 - Always overwrites the file
 - **'a'** for append
 - Writes at the end
 - **'r+'** or **'a+'** for updating (read + write/append)

Examples

- **f1 = open("myfile.txt")**
same as
f1 = open("myfile.txt", "r")
- **f2 = open("test\\sample.txt", "r")**
- **f3 = open("test/sample.txt", "r")**
- **f4 = open("D:\\piete\\Documents\\myfile.txt")**

Reading a file

- Three ways:
 - Global reads
 - Line by line
 - Pickled files

Global reads

- **fh.read()**
 - Returns **whole contents** of file specified by file handle **fh**
 - File contents are stored in a **single string** that might be very large

global_read.py

```
f2 = open("sequence.txt", "r")
bigstring = f2.read()
print(bigstring)
f2.close() # not required
```



```
> python3 global_read.py
>NC_000020
# sequence on the next 3
lines...
GGCCATGGTCAGCGTGAACG
CGCCCCTCGGGGCTCCAGTG
GAGAGTTCTTACGGTAAGTG
```

Line-by-line reads

line_read.py

```
f3 = open("sequence.txt", "r")
for line in f3 : # do not forget the column
    print(line)
f3.close() # not required
```



```
> python3 line_read.py
>NC_000020

# sequence on the next 3 lines...

GGCCATGGTCAGCGTGAACG

CGCCCCTCGGGGCTCCAGTG

GAGAGTTCTTACGGTAAGTG
```

What?

With one or more ***extra blank lines***

Why?

1. Each line ends with an end-of-line marker
2. **print(...)** *adds* an extra end-of-line

Exercise

Find a way to remove blank lines without using strip() functions...

```
> python3 line_read.py
>NC_000020

# sequence on the next 3 lines...

GGCCATGGTCAGCGTGAACG

CGCCCCTCGGGGCTCCAGTG

GAGAGTTCTTACGGTAAGTG
```



```
> python3 remove_blank_lines.py
>NC_000020
# sequence on the next 3 lines...
GGCCATGGTCAGCGTGAACG
CGCCCCTCGGGGCTCCAGTG
GAGAGTTCTTACGGTAAGTG
```

Making sense of file contents

- Most files contain more than one data item per line:

John,Doe,120 jefferson st.,Riverside, NJ, 08075

Jack,McGinnis,220 hobo Av.,Phila, PA,09119

- Must split lines
- **mystring.split(sepchar)**
where **sepchar** is a separation character
- returns a list of items

Splitting strings

```
>>> text = "Once upon a time in a far galaxy"  
>>> text.split()  
['Once', 'upon', 'a', 'time', 'in', 'a', 'far', 'galaxy']
```

```
>>> record = "1, 'Einstein, Albert', 1905, 1955"  
>>> record.split()  
["1, 'Einstein,", " 'Albert',", '1905,', '1955']
```

Not what we wanted!

Example

how2split.py

```
f5 = open("sample.txt", "r")
for line in f5 :
    words = line.split()
    for each_word in words:
        print(each_word)
f5.close() # not required
```



```
> python3 how2split.py
To
be
or
not
to
be
that
is
the
question
Now
is
the
winter
of
our
discontent
```

Other separators (I)

- **Commas**

- CSV Excel format
 - Values are separated by commas
 - Strings are stored without quotes
 - Unless they contain a comma
 - “Doe, Jane”, freshman, 90, 90
 - Quotes within strings are doubled

Other separators (II)

Tabs('\t')

Advantages:

Your fields will appear nicely aligned

Spaces, commas, ... are not an issue

Disadvantage:

You do not see them

They look like spaces

Why is it important?

- When you must pick your file format, you should decide how the data inside the file will be used:
 - People will read them
 - Other programs will use them
 - Will be used by people and machines

An exercise

- Converting our output to CSV format
 - Replacing tabs by commas
 - Easy
 - Will use string replace function

First attempt

tab2csv.py

```
fh_in = open('grades.txt', 'r') # the 'r' is optional
buffer = fh_in.read()
newbuffer = buffer.replace('\t', ',')
fh_out = open('grades0.csv', 'w')
fh_out.write(newbuffer)
fh_in.close()
fh_out.close()
print('Done!')
```



> python tab2csv.py
Done!

grades.txt

Alice	90	90	90	90	90
Bob	85	85	85	85	85
Carol	75	75	75	75	75



grades0.csv

Alice	90	90	90	90	90
Bob	85	85	85	85	85
Carol	75	75	75	75	75

Dealing with commas (I)

- Work line by line
- For each line
 - split input into fields using TAB as separator
 - store fields into a list
 - Alice 90 90 90 90 90
becomes
['Alice', '90', '90', '90', '90', '90']

Dealing with commas (II)

- Put within double quotes any entry containing one or more commas
- Output list entries separated by commas
 - **["Baker, Alice", 90, 90, 90, 90, 90]**
becomes
"Baker, Alice",90,90,90,90,90

Dealing with commas (III)

- Our troubles are not over:
 - Must store somewhere all lines until we are done
 - Store them in a list

Dealing with double quotes

- Before wrapping items with commas with double quotes replace
 - All double quotes by pairs of double quotes
 - **'Aguirre, "Lalo" Eduardo'**
becomes
'Aguirre, ""Lalo"" Eduardo'
then
""Aguirre, ""Lalo"" Eduardo""

General organization (I)

- `linelist = []`
- `for line in file`
 - `itemlist = line.split(...)`
 - `linestring = " # empty string`
 - `for each item in itemlist`
 - remove any trailing newline
 - double all double quotes
 - if item contains comma, wrap
 - add to `linestring`

General organization (II)

for line in file

...

for each item in itemlist

double all double quotes

if item contains comma, wrap

add to linestring

append linestring to stringlist

General organization (III)

- for line in file
- ...
 - remove last comma of linestring
 - add newline at end of linestring
 - append linestring to stringlist
- for linestring in in stringlist
 - write linestring into output file

grades.txt

```
Alice 90 90 90 90 90
Bob 85 85 85 85 85
Carol 75 75 75 75 75
Doe, Jane 90 90 90 80 70
Fulano, Eduardo "Lalo" 90 90 90 90
```

```
> python betterconvert2csv.py
```



great.txt

```
Alice,90,90,90,90,90
Bob,85,85,85,85,85
Carol ,75,75,75,75,75
"Doe, Jane",90,90 ,90 ,80 ,70
"Fulano, Eduardo ""Lalo""",90,90,90,90
```

Mistakes being made (I)

- **Mixing lists and strings:**
 - Earlier draft of program declared
 - **Linestring = []**
and did
 - **linestring.append(item)**
 - **Outcome was**
 - **['Alice,', '90,', ...]**
instead of
 - **'Alice,90, ...'**

Mistakes being made (II)

- **Forgetting to add a newline**
 - Output was a single line
- **Doing the append inside the inner loop:**
 - Output was
 - Alice,90
 - Alice,90,90
 - Alice,90,90,90
 - ...

Mistakes being made (III)

- **Forgetting that strings are immutable:**
 - Trying to do
 - `linestring[-1] = '\n'`
instead of
 - `linestring = linestring[:-1] + '\n'`
 - **Bigger issue:**
 - Do we have to remove the last comma?

Pickled files



- **import pickle**

- Provides a way to save complex data structures in a file
- Sometimes said to provide a ***serialized representation*** of Python objects

Basic primitives (I)

- **dump(object,fh)**
 - appends a sequential representation of **object** into file with file handle **fh**
 - **object** is virtually any Python object
 - **fh** is the handle of a file that must have been opened in **'wb'** mode



b is a special option allowing to write or read binary data

Basic primitives (II)

- **target = load(filehandle)**
 - assigns to **target** next pickled object stored in file **filehandle**
 - **target** is virtually any Python object
 - **filehandle** is filehandle of a file that was opened in **rb** mode

Example

```
>>> mylist = [ 2, 'Apples', 5, 'Oranges']
>>> mylist
[2, 'Apples', 5, 'Oranges']
>>> fh = open('testfile.txt', 'wb')
>>> import pickle
>>> pickle.dump(mylist, fh)
>>> fh.close()
>>> fhh = open('testfile.txt', 'rb')
>>> theirlist = pickle.load(fhh)
>>> theirlist
[2, 'Apples', 5, 'Oranges']
>>> theirlist==mylist
True
```

What was stored in testfile?

- Some binary data containing the strings 'Apples' and 'Oranges'

```
> vi testfile.txt
```



```
<80>^C]q^@(K^BX^F^@^@^@Applesq^AK^EX^G^@^@^@Orangesq^Be.
```

Using ASCII format

- When we need a pickled representation of objects that only contains printable characters
 - Must specify **protocol = 0**
- ***Advantage:***
 - Easier to debug
- ***Disadvantage:***
 - Takes more space

Example

```
>>> import pickle
>>> mydict = {'Alice': 22, 'Bob' : 27}
>>> fh = open('asciifile.txt', 'wb')
>>> pickle.dump(mydict, fh, protocol =
0)
>>> fh.close()
>>> fhh = open('asciifile.txt', 'rb')
>>> theirdict = pickle.load(fhh)
>>> print(mydict)
{'Alice': 22, 'Bob': 27}
>>> print(theirdict)
{'Alice': 22, 'Bob': 27}
```

What was stored in asciifile.txt?

```
> vi asciifile.txt
```



```
(dp0  
VAlice  
p1  
l22  
sVBob  
p2  
l27  
s.
```

Dumping multiple objects (I)

pickle_multi.py

```
import pickle
fh = open('asciifile.txt', 'wb')
for k in range(3, 6) :
    mylist = [i for i in range(1,k)]
    print(mylist)
    pickle.dump(mylist, fh, protocol = 0)
fh.close()

fhh = open('asciifile.txt', 'rb')
lists = [ ]      # initializing list of lists
while 1 :
    try:
        lists.append(pickle.load(fhh))
    except EOFError :
        break
fhh.close()
print(lists)
```



```
> python3 pickle_multi.py
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[[1, 2], [1, 2, 3], [1, 2, 3, 4]]
```


Dumping multiple objects (II)

- Note the way we test for end-of-file (**EOF**)
 - **while 1 : # means forever**
 try:
 lists.append(pickle.load(fhh))
 except EOFError :
 break

What is inside asciifile.txt?

```
> vi asciifile.txt
```



```
(lp0  
l1  
a12  
a.(lp0  
l1  
a12  
a13  
a.(lp0  
l1  
a12  
a13  
a14  
a.
```

Practical considerations

- You rarely pick the format of your input files
 - *May have to do **format conversion***
- You often have to use specific formats for your output files
 - *Often dictated by program that will use them*
- Otherwise ***stick with pickled files!***

File IO: opening and reading a file

I. Linewise iteration over file

Returns file handler

File mode (r, w, a, ...)

```
f = open('sequence.txt', 'r')
for line in f:
    if not line.startswith('#'):
        print(line)

f.close()
```

Loop variable

>NC_000020
GGCCATGGTCAGCGTGAACG
CGCCCCTCGGGGCTCCAGTG
GAGAGTTCTTACGGTAAGTG

II. Shorter and better

```
with open('sequence.txt', 'r') as f:
    for line in f:
        if not line.startswith('#'):
            print( line.rstrip() )
```

>NC_000020
GGCCATGGTCAGCGTGAACG
CGCCCCTCGGGGCTCCAGTG
GAGAGTTCTTACGGTAAGTG

Example: FASTA format I

```
with open('fly3utr.txt', 'r') as f:
    for line in f:
        if line.startswith('>'):
            print(line.rstrip())
```



```
>CG11604
>CG11455
>CG11488
```


```
fly3utr.txt:
>CG11604
TAGTTATAGCGTGAGTTAGT
TGTAAGGAACGTGAAAGAT
AAATACATTTTCAATACC
>CG11455
TAGACGGAGACCCGTTTTTC
TTGGTTAGTTTCACATTGTA
AAACTGCAAATTGTGTAAAA
ATAAAATGAGAAACAATTCT
GGT
>CG11488
TAGAAGTCAAAAAAGTCAAG
TTTGTTATATAACAAGAAAT
CAAAAATTATATAATTGTTT
TTCACCTCT
```

What if we want to show
the length of sequence
for each record?

Example: FASTA format II

```
name = ''
with open('fly3utr.txt', 'r') as f:
    for line in f:
        line = line.rstrip()
        if line.startswith('>'):
            if name: # Empty str is False
                print(name, length)
            name = line[1:]
            length = 0
        else:
            length += len(line)
    print(name, length)
```

```
>CG11604
TAGTTATAGCGTGAGTTAGT
TGTAAGGAACGTGAAAGAT
AAATACATTTTCAATACC
>CG11455
TAGACGGAGACCCGTTTTTC
TTGGTTAGTTTCACATTGTA
AACTGCAAATTGTGTAAAA
ATAAAATGAGAAACAATTCT
GGT
>CG11488
TAGAAGTCAAAAAAGTCAAG
TTTGTTATATAACAAGAAAT
CAAAAATTATATAATTGTTT
TTCACTCT
```



CG11604	58
CG11455	83
CG11488	68

File Input/Output: Redirection to a file

The **print** command sends its argument (the string in parentheses) to **standard output**, which is normally the terminal.

A simple way to send program output to a file (instead of printing it on the screen) is to use the Unix **redirection** sign ">".

Example:

To print the results of the **test.py** program to a file named **test.out**, use the following Unix command:

```
python test.py > test.out
```

Check the content of 'test.out'

Exercise 1 – Reading and writing files

You will be provided with a raw microarray data file called '**raw_data.txt**'

This file contains 6 columns of information:

- Probe
- Name
- Chromosome
- Position
- Feature
- Sample A data
- Sample B data

You should write a program to filter this data.

The first line is a header and should be kept.

For each other line calculate the log2 of Sample A data and Sample B data and keep the line only if:

- Log2 of either Sample A or B is greater than 2
- The log2 difference (either positive or negative) between Sample A and B is greater than 3 (ie an 8 fold change in raw value)

Print the filtered results in a file called '**filtered_data.txt**'

Exercise 2 – Reading and writing files

You will be provided with two files.

Annotation.txt contains a list of sequence accession codes and their associated descriptions, separated by tabs.

Data.txt has the same list of accessions (though not in the same order) alongside some tab separated data values.

You should combine these files to produce a single file containing the accession, data and description for each gene. Your script should perform basic sanity checks on the data it reads (eg checking that you have both an accession and description for each gene, and checking that each accession in the data file really does have annotation associated with it before printing it out).

Summary

- Strings, lists, iterators, and tuples are all **sequences**
- **Lists** for storage of element of equal elements
 - More flexible, more memory consumption
- **Tuples** for storage of different elements
 - Immutable, less memory consumption
- **Iterators** for fast iteration
 - Least memory consumption, can be only used once!
- Often, a **list comprehension** can replace a for loop with an if-construction
- Convert strings into lists and vice versa with join and split
- File object provides line-wise iteration

Functions

Functions

- Often, self-contained tasks occur in many different places
 - ⇒ we may want to separate their description from the rest of our program.
- Code for such a task is called a ***function***
- Examples of such tasks:
 - ⇒ reverse complementing a sequence
 - ⇒ filtering out all negative numbers from a list

Function Syntax

Syntax

```
def <functionname> (<arg1>, <arg2>, ...):  
    <block>  
    return <something>
```

Example

```
def sum_up_numbers (num1, num2):  
    my_sum = num1 + num2  
    return my_sum
```

Calling a function

Function definition

```
def sum_up_numbers (num1, num2):  
    my_sum = num1 + num2  
    return my_sum
```

Function calls

sum_up_numbers (1, 5)	→	6
sum_up_numbers (num1=1, num2=5)	→	6

Example: Maximum element of a list

Function to find the largest entry in a list

```
def find_max(data):  
    max = data.pop()  
    for x in data:  
        if x > max:  
            max = x  
    return max
```

```
data = [1, 5, 1, 12, 3, 4, 6]  
print("Data:", data)  
print("Maximum: %i" %find_max(data))
```

← Function declaration

} Function body

← Function result

← Function call



```
> python3 find_max.py  
Data: [1, 5, 1, 12, 3, 4, 6]  
Maximum: 12
```

Lambda Functions

- Kind of anonymous functions
- Equivalent to normal functions
 - ⇒ Not bound to a name
 - ⇒ Different syntax

normal_function.py

```
def f(x):  
    return (x-3)**2  
  
print( f(5) )
```



```
> python3 normal_function.py  
4
```

lambda_function.py

```
f = lambda x: (x-3)**2  
  
print( f( 5 ) )
```



```
> python3 lambda_function.py  
4
```


Examples

```
>>> map(lambda x: x*3, [1,2,3])  
<map object at 0x7f920f05cc50>
```

```
>>> list(map(lambda x: x*3, [1,2,3]))  
[3, 6, 9]
```

```
>>> list(filter(lambda x: x>=1.0, [1.2,0.5,0.7,1.3]))  
[1.2, 1.3]  
>>> list(filter(lambda x: x!=0, map(lambda x: x-2, [4,2,5])))  
[2, 3]
```

```
>>> from functools import reduce  
>>> reduce(lambda x,y: x+y if x<=2 else x*y, (1,2,3))  
9
```

'map.py'

```
print(map(lambda x: x*3, [1,2,3]))  
print(filter(lambda x: x>=1.0, [1.2,0.5,0.7,1.3]))  
print(filter(lambda x: x!=0, map(lambda x: x-2, [4,2,5])))  
print(reduce(lambda x,y: x+y if x<=2 else x*y, (1,2,3)))
```



```
> python map.py  
[3, 6, 9]  
[1.2, 1.3]  
[2, 3]  
9
```

Built-in functions

- **print(...)** is always available
- Other functions are parts of ***modules***
 - **sqrt(...)** is part of **math** module
- Before using any of these functions we must **import them**
 - **from math import sqrt**
 - **from random import randint, uniform**



Note the comma!

More about modules

- We can write our own modules
 - Can be situations where two or more modules have functions with the same names
 - Solution is to **import** the modules
 - **import math**
- Can now use all functions in module
 - Must **prefix** them with module name **math.sqrt(...)**

Your two choices

- When you want to use the function **sqrt()** from the module **math**, you can either use
 - **from math import sqrt**
and refer directly to **sqrt()**
 - **import math**
and refer to the function as **math.sqrt()**

Good practice rules

- Put all your **import** and **use** statements at the beginning of your program
 - Makes the program *more legible*
- As soon as you use several modules, avoid **import from**
 - Easier to find which function comes from which module

Writing your own function

- Very easy
- Write
 - **def** *function_name(parameters)* :
 statements
 return *result*
- Observe the column and the indentation

What it does



Example

```
>>> def maximum (a, b) :  
...         if a >= b :  
...             max = a  
...         else :  
...             max = b  
...         return max  
...  
>>> maximum(5,6)  
6  
>>> maximum(2.0,3)  
3  
>>> maximum("big", "tall")  
'tall'  
>>> maximum('big', 'small')  
'small'  
>>> maximum ('a', 3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 2, in maximum  
TypeError: '>=' not supported between instances of 'str'  
and 'int'
```

Does not work: unorderable types: str() >= int()

Multiple return statements

```
>>> def maximum2 (a, b) :  
...         if a >= b :  
...             return a  
...         else :  
...             return b  
...  
>>> maximum2(0, -1)  
0
```

No return statement

```
>>> def goodbye() :  
...     input('Hit return when you are done.')  
...  
...  
>>> goodbye()  
Hit return when you are done.  
>>> goodbye  
<function goodbye at 0x7f428239a1e0>
```

These pesky little details...

- The first line of the function declaration
 - Starts with the keyword **def**
 - Ends with a **colon**
- Don't forget the parentheses when you call a function
 - **goodbye()**



```
# firstfunctions.py

"""
This program contains two functions that convert C into F and F into C
"""

def celsius (temperature) :
    return (temperature - 32)*5/9

def fahrenheit (temperature) :
    return (temperature*9/5 + 32)

degrees = float(input('Enter a temperature: '))

print('%.1f Fahrenheit is same as' % degrees + ' %.1f Celsius' %
celsius(degrees))
print('%.1f Celsius is same as' % degrees + ' %.1f Fahrenheit' %
fahrenheit(degrees))
input('Hit return when you are done')
```

```
> python firstfunctions.py
Enter a temperature: 37
37.0 Fahrenheit is same as 2.8 Celsius
37.0 Celsius is same as 98.6 Fahrenheit
Hit return when you are done
```

Creating a module

- Put the functions in a separate file

```
#twofunctions.py
""" This module contains two functions
"""
def celsius (temperature) :
    return (temperature - 32)*5/9
def fahrenheit (temperature) :
    return (temperature*9/5 + 32)
```

Using a module

```
#samefunctions.py

""" This program calls two functions.
"""

from twofunctions import celsius, fahrenheit

degrees = float(input('Enter a temperature: '))
print('%.1f Fahrenheit is same as' % degrees + ' %.1f Celsius' % celsius(degrees))
print('%.1f Celsius is same as' % degrees + ' %.1f Fahrenheit' %
fahrenheit(degrees))
input('Hit return when you are done')
```



```
> python samefunctions.py
Enter a temperature: 37
37.0 Fahrenheit is same as 2.8 Celsius
37.0 Celsius is same as 98.6 Fahrenheit
Hit return when you are done
```

Notes

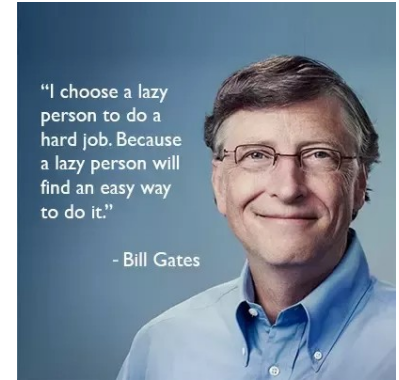
- Module name must have **.py** suffix
- **import** statement should contain module name **stripped of that suffix**

Parameters w/ default values

```
>>> def aftertax( price, taxrate = 0.0825) :  
...         return price*(1 + taxrate)  
...  
>>> aftertax(100)  
108.25  
>>> aftertax(100,0)  
100  
>>> aftertax(100, .12)  
112.000000000000001
```


Why you should write functions

- Makes your code more readable
 - Hides the details
 - Keeps each piece of code shorter
- Allows you to reuse your work



Example – scope issues (I)

- Function titlecase:
 - Converts a string to “Title Case” format
 - Same as MS Word
 - Words in the string will
 - Start with an upper case letter
 - All other letters will be lower case
 - No special handling of articles and short prepositions

Example – scope issues (II)

```
def titlecase (instring) :  
    # converts instring into Title Case  
    stringlist = instring.split(" ")  
    outstring = "" # empty string  
    for item in stringlist :  
        newitem = item[0].upper() + item[1:].lower()  
        outstring+= newitem + " "  
    if outstring[-1] == " " :  
        outstring = outstring[0:-1]  
    return outstring  
  
print( titlecase("what happens in gent stays in ghent") )
```

What if the variable names

- ***stringlist***
- ***outstring***
- ***item***
- ***newitem***

were used elsewhere in the program?

What would happen?

The answer

- ***Nothing***

**What happens in Ghent,
stays in Ghent**

**What happens inside a function
stays inside the function**

How?

- Python creates new instances of
 - *stringlist*
 - *outstring*
 - *item*
 - *newitem*
- that only exist inside the function

Example

donothing.py

```
def donothing() :  
    k = 5  
  
k = 0  
donothing()  
  
print("k = " + str(k))
```



```
> python donothing.py  
k = 0
```

The exception

**What happens in Ghent,
stays in Ghent UNLESS you post it
on Twitter**



**What happens inside a function stays inside
the function UNLESS
you specify the variable is global**

Example

dosomething.py

```
def dosomething() :  
    global k  
    k = 5  
  
k = 0  
dosomething()  
  
print("k = " + str(k))
```



```
> python dosomething.py  
k = 5
```


Advantages

- Some variables are inherently global
 - Useful in simulations of complex systems
 - You can pass “invisible parameters” to a function
- *Some people will think you are too lazy to pass them as parameters*

Disadvantages

- Updating variables “secretly” is very dangerous
- Can—and often will—confuse the reader
- Will cause hard to find errors

The verdict

- Do not use global variables in your functions unless you have a good reason to do so
- ***Python default option is the right one:***

**What happens inside a function
stays inside the function**

Some useful Python modules

Separate libraries of code that provide specific functionality for a certain set of tasks

Some base Python modules:

- **os** **and** **shutil**
 - ◊ Useful for interacting with the **o**perating **s**ystem
- **sys**
 - ◊ Useful for interacting with the Python interpreter
- **subprocess**
 - ◊ Useful for calling external software from your Python script
- **re**
 - ◊ Regular expressions

Loading modules in a script


Use the import command at the **top of your script:**

```
import os
import os as opsys
from os import *
from os import <function/submodule>
```

Loading modules in a script


Use the **import** command at the ***top*** of your script:

```
import os  
import os as opsys
```



use as `os.function_name()`
`opsys.function_name()`

```
from os import *  
from os import <function/submodule>
```



use as
`function_name()`

The os/shutil modules

Functions provide UNIX commands

os/shutil function	UNIX equivalent
os.remove("filename")	rm filename
os.rmdir("directory")	rm -r directory
os.chdir("directory")	cd directory
os.listdir("directory")	ls directory
os.mkdir("directory")	mkdir directory
shutil.copy("oldfile", "newfile")	cp oldfile newfile
shutil.move("oldfile", "newfile")	mv oldfile newfile

Looping over files with os.listdir

os_listdir.py

```
import os

# Current directory
directory = "./"

# Obtain list of files in directory
files = os.listdir(directory)

# Loop over files that end with .txt
for file in files:
    if file.endswith(".txt"):
        f = open(directory + file, "r")
        # print the first line of each file
        print( f.readline().rstrip() )
        f.close()
```



```
> python3 os_listdir.py
>NC_000020
>CG11604
```


The sys module

Useful:

`sys.path`

`sys.exit()`

`sys.argv`

Using sys.path

sys.path is a list of directories in your PYTHONPATH

PYTHONPATH is an environment variable which you can set to add additional directories where python will look for modules and packages.

```
import sys
import os

cwd = os.getcwd()

print( 'BEFORE append...' )
print( str.join('\n', sys.path ) )

print( 'AFTER append...' )
# Add directories with append!
sys.path.append( cwd )
print( str.join('\n', sys.path ) )
```



```
> python3 sys_path.py
BEFORE append...
/data1/ownCloud/PROG4BIO_2017/PY4BIO/Lecture2
/usr/lib64/python37.zip
/usr/lib64/python3.7
/usr/lib64/python3.7/lib-dynload
/home/pieterdb/.local/lib/python3.7/site-packages
/usr/local/lib64/python3.7/site-packages
/usr/local/lib/python3.7/site-packages
/usr/lib64/python3.7/site-packages
/usr/lib/python3.7/site-packages
AFTER append...
/data1/ownCloud/PROG4BIO_2017/PY4BIO/Lecture2
/usr/lib64/python37.zip
/usr/lib64/python3.7
/usr/lib64/python3.7/lib-dynload
/home/pieterdb/.local/lib/python3.7/site-packages
/usr/local/lib64/python3.7/site-packages
/usr/local/lib/python3.7/site-packages
/usr/lib64/python3.7/site-packages
/usr/lib/python3.7/site-packages
/data1/ownCloud/PROG4BIO_2017/PY4BIO/Lecture2
```

Using sys.exit()

sys.exit () will immediately stop the interpreter and exit out of the script

```
import sys
import random

something_important = random.choice([True, False])

if something_important == False:
    print( "Oh no, something is wrong!!!")
    sys.exit()
else:
    while True:
        print("Live is fine...")
```



```
> python3 sys_exit.py
Oh no, something is wrong!!!
```

Or...
Ctrl-C

Processing command-line arguments

cli1.py

```
#!/usr/bin/env python3

a = 2
b = 3
result = a + b

print('The result is ', result)
```

chmod +x on a file means, that you'll make it executable.

```
> chmod +x cli1.py
> ./cli1.py
The result is 5
```

What if we want to run this script repeatedly with different values for variables a and b?

Using sys.argv

sys.argv is a list of command-line input arguments
Always read as **strings**!

```
sys.argv[0] ## The name of the script  
sys.argv[1] ## The value of the first command line arg  
sys.argv[2] ## The value of the second command line arg  
...
```

sys.argv script - v1

sys_arg_v1.py

```
import sys

value = sys.argv[1]
print("You provided", value)
```

Calling script from console **with** an argument

> python3 sys_arg_v1.py 123456
You provided 123456

Calling script from console **without** an argument

> python3 sys_arg_v1.py
Traceback (most recent call last):
 File "sys_arg_v1.py", line 3, in <module>
 value = sys.argv[1]
IndexError: list index out of range

error if no argument is provided!!!

sys.argv script - v2

sys_arg_v2.py

```
import sys
import argparse

assert(len(sys.argv) == 2), " Give me an argument!"
value = sys.argv[1]
print("You provided", value)
```

Calling script from console **with** an argument

```
> python3 sys_arg_v2.py 123456
You provided 123456
```

Calling script from console **without** an argument

```
> python3 sys_arg_v2.py
Traceback (most recent call last):
  File "sys_arg_v2.py", line 4, in <module>
    assert(len(sys.argv) == 2), " Give me an argument!"
AssertionError: Give me an argument!
```

A more useful error is thrown if no argument is provided!!!

Using sys.argv – type casting

```
import sys

assert(len(sys.argv) == 2), "Expected an argument"
value = sys.argv[1]
print(value + 25)
```



```
> python3 sys_arg_type_casting_1.py 75
Traceback (most recent call last):
  File "sys_arg_type_casting_1.py", line 5, in
<module>
    print(value + 25)
TypeError: can only concatenate str (not
"int") to str
```

```
import sys

assert(len(sys.argv) == 2), "Expected an argument"
value = sys.argv[1]
print(float(value) + 25)
```



```
> python3 sys_arg_type_casting_2.py 75
100.0
```

Key Points to Remember:

- 1) Type Conversion is the conversion of object from one data type to another data type.
- 2) Implicit Type Conversion is automatically performed by the Python interpreter.
- 3) Python avoids the loss of data in Implicit Type Conversion.
- 4) Explicit Type Conversion is also called Type Casting, the data types of object are converted using predefined function by user.
- 5) In Type Casting loss of data may occur as we enforce the object to specific data type.

The try... except pair (I)

- **try:**
 <statements being tried>
except Exception as ex:
 <statements catching the exception>
- Observe
 - the colons
 - the indentation

The try... except pair (II)

- **try:**
 <statements being tried>
except Exception as ex:
 <statements catching the exception>
- If an exception occurs while the program executes the statements between the **try** and the **except**, control is **immediately transferred** to the **statements after the except**

Using sys.argv – try/except

sys_arg_try_except.py

```
import sys

if len(sys.argv) != 2:
    sys.stderr.write("USAGE: python3 %s < value >\n" % sys.argv[0])
    sys.exit(1)

value = sys.argv[1]

try:
    value = float(value)
except:
    raise AssertionError("Couldn't make the input a float!")

print(value + 25)
```

Calling script from console **without** an argument

```
> python3 sys_arg_try_except.py
USAGE: python3 sys_arg_try_except.py < value >
```

Calling script from console **with** acceptable argument

```
> python3 sys_arg_try_except.py 75
100.0
```

Calling script from console **with** 'stupid' argument

```
> python3 sys_arg_try_except.py Jared
Traceback (most recent call last):
  File "sys_arg_try_except.py", line 10, in <module>
    value = float(value)
ValueError: could not convert string to float: 'Jared'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "sys_arg_try_except.py", line 12, in <module>
    raise AssertionError("Couldn't make the input a float!")
AssertionError: Couldn't make the input a float!
```

Interesting science libraries (but far from complete...)

- **scipy and numpy**
 - Work with matrices
 - Fundamental scientific computing
 - Matlab in Python
 - <https://www.scipy.org/>
 - <http://www.numpy.org/>
- **pandas**
 - Data structures (R for python ish)
 - <https://pandas.pydata.org/>
- **scikit-learn**
 - Machine learning
 - <http://scikit-learn.org/stable/>

Exercise

Write a script that prints the sum of an arbitrary number of values given at the command line

Desired output:

No arguments given:

```
> python3 sys_arg_sum_cli.py  
USAGE: python3 sys_arg_sum_cli.py < value 1 > < value 2 > ... < value n >
```

With arguments...

```
> python3 sys_arg_sum_cli.py 11 12 23 44 55  
values = [11.0, 12.0, 23.0, 44.0, 55.0]  
sum = 145.0
```

Pattern matching

A very sophisticated kind of logical test is to ask whether a string contains a ***pattern***
e.g. does a yeast promoter sequence contain the MCB binding site, ACGCGT?

```
name = 'YBR007C'  
dna = 'TAATAAAAAACGCGTTGTCG'  
if 'ACGCGT' in dna:  
    print('%s has MCB!' % name)
```

20 bases upstream of the yeast gene YBR007C

YBR007C has
MCB!

The membership operator **in**

The pattern for the MCB binding site

Regular expressions

We already defined a simple pattern: `ACGCGT`

What if we don't care about the 3rd position?

`ACGCGT` `ACCGT` `ACACGT` `ACTCGT`

- Python provides a pattern-matching engine
- Patterns are called regular expressions
- They are extremely powerful
- Often called "regexps" for short
- module `re`

Motivation: N-glycosylation motif

- Common post-translational modification
- Attachment of a sugar group
- Occurs at asparagine residues with the consensus sequence NX₁X₂, where
 - X₁ can be anything (but proline inhibits)
 - X₂ is serine or threonine
- Can we detect potential N-glycosylation sites in a protein sequence?

Building regexps I: Character Groups

- In general square brackets denote a set of alternative possibilities
 - E.g. **[abc]** -> matches a,b, or c
- Use - to match a range of characters: **[A-Z]**
- Negation : **[^X]** matches anything but X
- **.** matches anything

Building regexps II: Abbreviations

- **\d** matches any decimal digits [0-9]
- **\D** matches any non-digit [^0-9]
- Equivalent syntax for ...
 - whitespaces (**\s** and **\S**)
 - alphanumeric (**\w** and **\W**)

Building regexps III: Repetitions

- Use `*` to match none or any number of times
 - E.g. `ca*t` matches: `ct`, `cat`, `caat`, `caaat`, `caaaat`, ...
- Use `+` to match one or any number of times
 - E.g. `ca+t` matches `cat`, `caat`, `caaat`, `caaaat`, ...
- Use `?` to match none or once
 - E.g. `bio-?info` matches `bioinfo` and `bio-info`
- Use `{m, n}` to specifically set the number of repetitions (min `m`, max `n`)
 - E.g. `ab{1, 3}c` will match `abc`, `abbc`, `abbbc`

Using regular expressions

- Compile a regular expression object (pattern) using `re.compile`
- pattern has a number of methods
 - `match` (in case of success returns a Match object, otherwise None)
 - `search` (scans through a string looking for a match)
 - `findall` (returns a list of all matches)

```
>>> import re
>>> pattern = re.compile('[ACGT]')
>>> if pattern.match("A"): print("Matched")
Matched
>>> if pattern.match("a"): print("Matched")
>>>
```

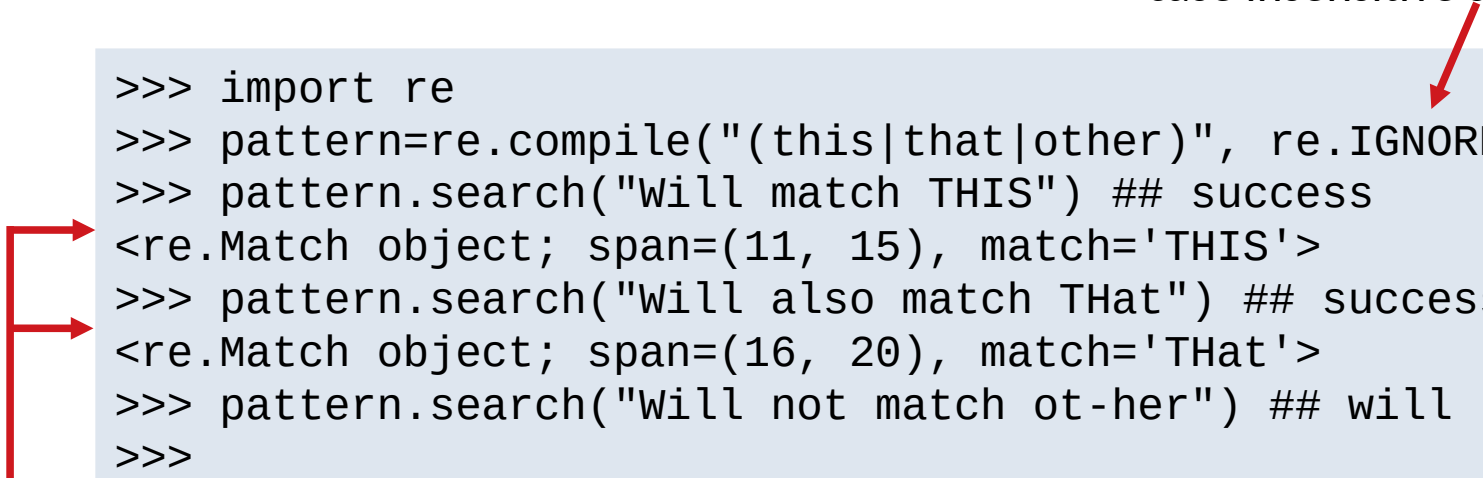
successful match

unsuccessful, returns None
by def. case sensitive

Matching alternative strings

- `/(this|that)/` matches "this" or "that"
- ...and is equivalent to: `/th(is|at)/`

case insensitive search pattern



```
>>> import re
>>> pattern=re.compile("(this|that|other)", re.IGNORECASE)
>>> pattern.search("Will match THIS") ## success
<re.Match object; span=(11, 15), match='THIS'>
>>> pattern.search("Will also match THat") ## success
<re.Match object; span=(16, 20), match='THat'>
>>> pattern.search("Will not match ot-her") ## will return None
>>>
```

Python returns a description of the match object

Word and string boundaries

`^` matches the start of a string

`$` matches the end of a string

`\b` matches word boundaries

“Escaping” special characters

- `\` is used to "escape" characters that otherwise have meaning in a regexp
- so `\[` matches the character "["
 - if not escaped, "[" signifies the start of a list of alternative characters, as in `[ACGT]`
 - All special characters: `. ^ $ * + ? { [] \ | ()`

Substitutions/Match Retrieval

- regexp methods can be used without compiling (less efficient but easier to use)

→ Example `re.sub` (substitution):

```
>>> re.sub("(red|blue|green)", "color", "blue socks and red shoes")  
'color socks and color shoes'
```

```
>>> e,raw,frm,to = re.findall("\d+", \  
... "E-value: 4, \  
... Raw Bit Score: 165, \  
... Match position: 362-419")  
>>> print(e, raw, frm, to)  
4 165 362 419
```

matches one or more digits

The result, a list of 4 strings,
is assigned to 4 variables

`\` allows multiple line commands
alternatively, construct multi-line
strings using triple quotes `""" ... """`

N-glycosylation site detector

```
import re

protein="MGMFFNLRSNIAKKKAMDNGLSLPISRNGSSNNIKDKRSEHNSNSLKGKYRYQPRSTPSKFQLTVSITSLI \
IIAVLSLYLFISFLSGMGIGVSTQNGRSLLGSSKSSSENYKTIDLEDEEYYDYDFEDIDPEVISKFDDGVQ \
HYLISQFGSEVLTPKDDEKYQRELNMLFDSTVEEYDLSNFEGAPNGLETRDHILLCIPLRNAADVLPMLF \
KHLMNLTYPHELIDLAFLVSDCSEGDTTLDALIAYSRHLQNGTLSQIFQEIDAVIDSQTKGTDKLYLKYM \
DEGYINRVHQAFSPPFHENYDKPFRSVQIFQKDFGQVIGQGFSRHAHVQVGIRRKLMGRARNWLTANAL \
KPYHSWVYWRDADVELCPGSVIQDLMSKNYDVI".upper().replace("\n", "")

for match in re.finditer("N[^P][ST]" + protein):
    print(match.group(), match.span())
```

`re.finditer`
provides an iterator
over match-objects

`N[^P][ST]`
- the main regular
expression

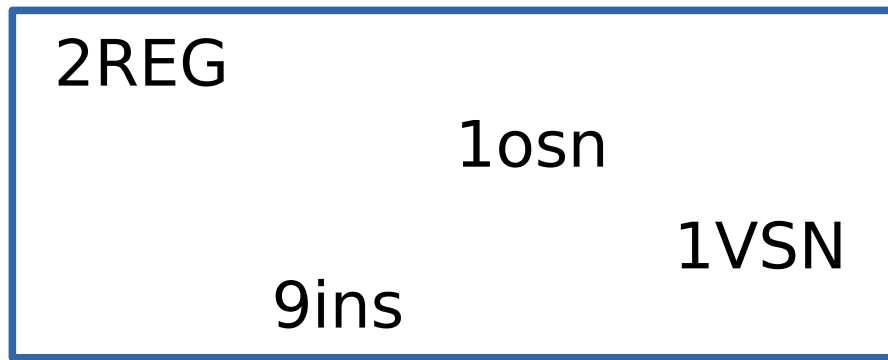
> python3 nglycosylation.py
NGS (26, 29)
NLT (217, 220)
NGT (253, 256)

`match.group` and `match.span` print the actual matched string and the position-tuple

Test your Regular Expressions

www.pythex.org

- Develop regular expressions
- Test them on examples of your choice



2REG
10sn
1VSN
9ins

PDB IDs

`([1-9][A-Za-z0-9]{3})`

Summary

- **Regular expression** as powerful tools to match patterns
- Allow matching of character groups, repetitions, alternatives, etc.
- Learn the meaning of special characters: . ^ \$ * + ? { [] \ | ()
- Python offers **regex** functions in the **re** module: match, search, findall, compile, etc.
- Regular expressions can be used to find motifs in sequences