

If you repeat more than once, then write a script to do it!



Part II is a basic introduction to programming

What is Python? Why Python?



- A relatively new (~90') high-level scripting language
- Python is an interpreted language: we write our code in a text file (*script*) and then the Python interpreter translates it to machine code for the CPU on the fly.
- Python2.x vs Python 3.x
- Why python?
 - Easy to use.
 - Widely spread in the bioinformatic community (has recently outranked Perl) as well as in many other fields.
 - Expressive (in this context means that a single line of Python code can do more than a single line of code in most other languages) and readable
 - Cross-platform
 - Free
 - Nature vol 518 (feb 2015) pp 125:

What matters most in the early stages is having a good support network. "Pick the programming language based on what people around you are using,".... Increasingly, that language is Python

- Why do we teach python? **Intro to programming**
- Anaconda is a distribution of Python for large-scale data processing, predictive analytics, and scientific computing



Anaconda

- A computer program is a detailed, step-by-step set of instructions telling a computer exactly what to do.
- The instructions are executed sequentially
- Programming languages are human readable notations to express computations in a exact non unambiguous way.
 - Programs are written in some high-level human-readable/writable code (the *source code*)
 - Then the instructions are translated (compiled) to a machine-readable code (the *compiled file* or *machine code*).
 - Note: the source code is independent on the type of computer and OS (platform independent) but the compiled file will only run in a specific machine/OS (platform-specific).
- Programming languages can be classified into two broad types, based on how and when the compilation of the human readable code (*source code*) into machine code is done:
 - Compiled languages--> compiled programs are stored as *binary files*
 - Interpreted languages (scripting languages)--> interpreted programs are *text files* (scripts)
- Each programming language has a specific:
 - Lexicon: the set of words that can be used in a program.
 - Syntax: valid ways of combining the lexicon elements.
 - Semantics: meaning of the syntactically correct expressions
 - **In spite of this, all programming languages have a common set of elements or structures.**

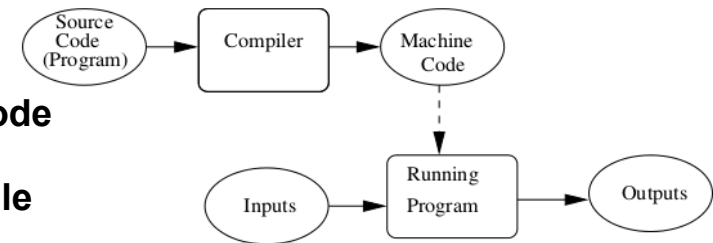


Figure 1.2: Compiling a High-Level Language

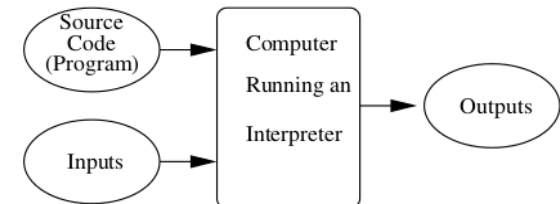
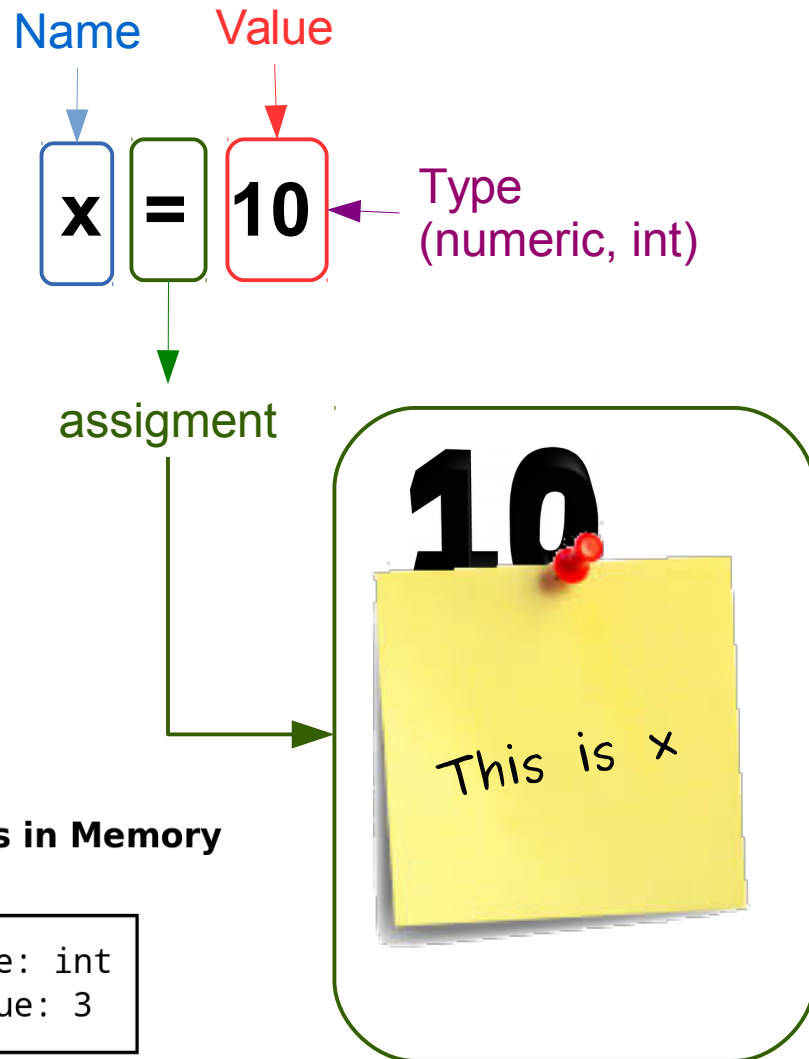


Figure 1.3: Interpreting a High-Level Language.

- What is a variable
- Variable attributes:
 - Name
 - Value
 - Type

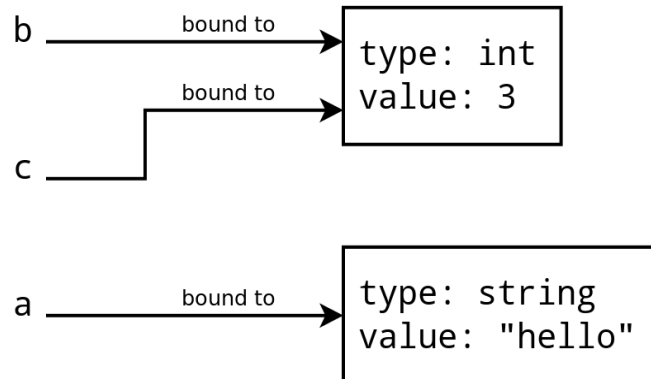


**Executed Code:
Variable Assignment**

```
a = 3
b = a
c = a
a = "hello"
```

Variables

Values in Memory



A) Python interactive interpreter

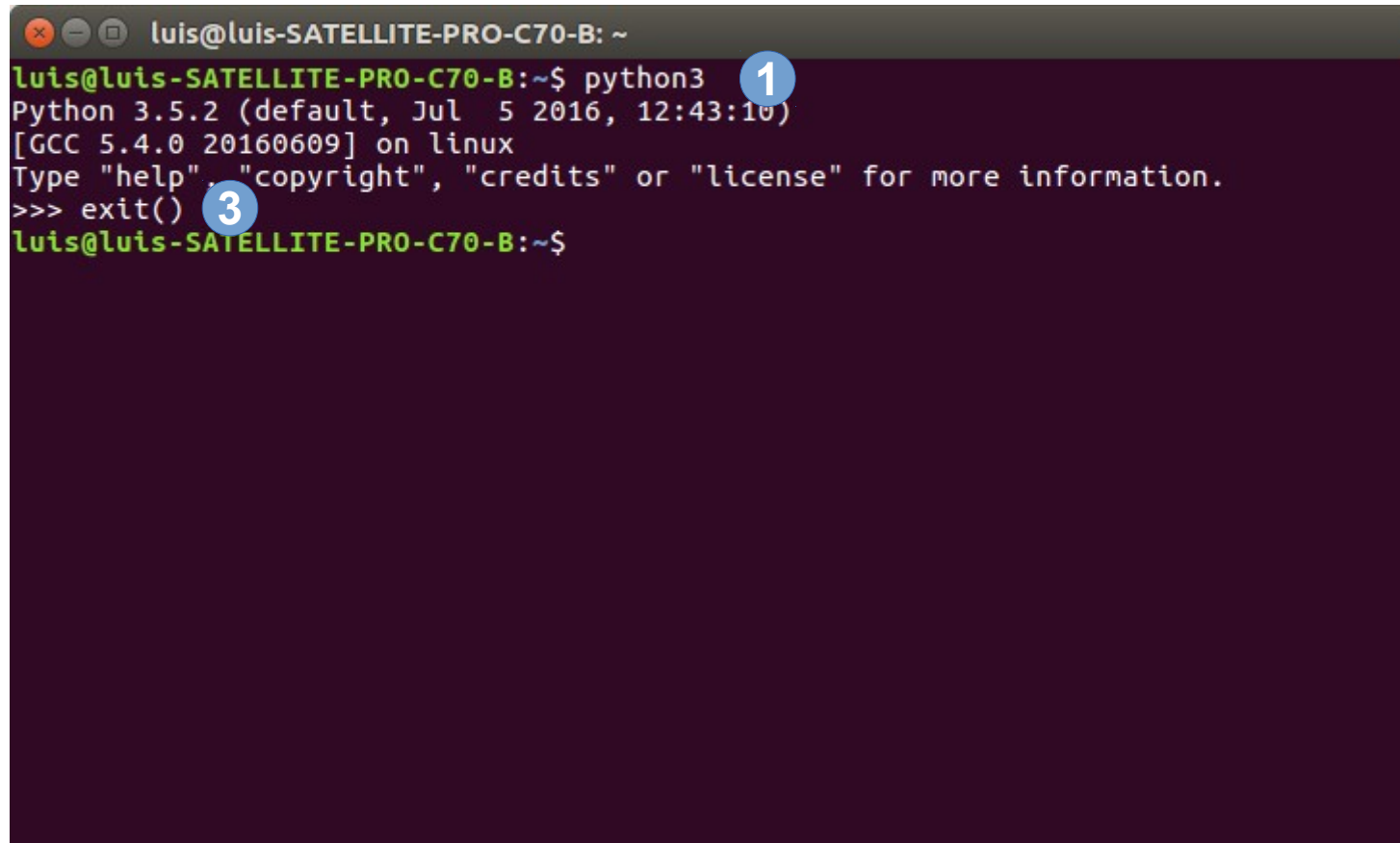
- 1) Open an interactive session with the python interpreter by typing "python3" in a terminal. Note that the prompt changes to ">>>" indicating we're now in a python shell.
- 2) Type the python commands to be executed (one statement at a time)
- 3) Exit the python shell by typing pressing ctrl key+D or typing:

>>> *exit()*

Note that all the code and results are now gone!

Terminal: text input/output environment through which you could interact with a computer

- On linux: Ctr+T
- On Mac OS: Command+space (search for "terminal")
- On Windows: Start→Search "powershell"




```
luis@luis-SATELLITE-PRO-C70-B: ~  
luis@luis-SATELLITE-PRO-C70-B:~$ python3 1  
Python 3.5.2 (default, Jul 5 2016, 12:43:10)  
[GCC 5.4.0 20160609] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> exit() 3  
luis@luis-SATELLITE-PRO-C70-B:~$
```

Basic types of variable:

- Integer number (*int*). Stores whole numbers (between -2147483648 and +2147483648) without fractional component
- Large or fractional number (*float*). Holds numbers that are extremely large or have decimal fractions.
- Text (*string*). store a sequence of text characters. They are defined using quotation marks (either single or double)
- Boolean. True or False (0 or 1, respectively).

In a Python shell type the following commands (then press enter key):

```
>>> x=5
>>> x
>>> y=132
>>> y
>>> x=y
>>> x
>>> MiTexto1='I Love Programming'
>>> MiTexto1
>>> PassHPBBM=true
>>> PassHPBBM=True
>>> PassHPBBM
>>> MiTexto2=I Love Programming
```



Basic data types

Basic types of variable:

- Integer number (*int*). Stores whole numbers (between -2147483648 and +2147483648) without fractional component
- Large or fractional number (*float*). Holds numbers that are extremely large or have decimal fractions.
- Text (*string*). store a sequence of text characters. They are defined using quotation marks (either single or double)
- Boolean. True or False (0 or 1, respectively).

Interactive Python session (in a python shell)
In the following slides I used the following color code:
What you type is in black
The computer output in blue

Error message. Python doesn't understand what "true" is

True is a "reserved word. It has meaning for python
Syntax is case-sensitive!

In a Python shell type the following commands (then press enter key):

```
>>> x=5
>>> x
5
>>> y=132
>>> y
132
>>> x=y
>>> x
132
>>> MiText01='I Love Programming'
>>> MiText01
I Love Programming
>>> PassHPBBM=true
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    passHPBBM=true
NameError: name 'true' is not defined
>>> PassHPBBM=True
>>> PassHPBBM
True
>>> MiText02=I Love Programming
SyntaxError: invalid syntax
```

Remember to press enter

Use quotation marks to define a string

Mathematical operators

Operator	Symbol	Action (numbers)	Action (strings)
Addition	+	Sum	Concatenate
Subtraction	-	Subtraction	none
Multiplication	*	Multiplication	Concatenate copies
Division	/	Division	none
Power	**	Power	none
Modulo	%	Remainder	none
Truncated division	//	Non-fractional part of result	none
Assignment	=	Assigns a number to variable	Assigns a string to variable

Comparative operators

Operator	Symbol
Equal to	==
Not equal to	!=, <>
Greater than	>
Less than	<
Greater or equal	>=
Less or equal	<=

Logical operators

Operator	Symbol
And	And, &(bitwise)
Or	or, (bitwise)
Not	Not, !

Do not mistake the assignment operator (=) for the equality operator (==)

Precedence. Python follows the same precedence rules for its mathematical operators that mathematics does: parentheses, exponentiation, Multiplication & Division, Addition and subtraction.

Type the following statements in an interactive python interpreter and hit enter (shown just in the first one)

IMPORTANT: before hitting enter, try to guess the result you will get.

```
>>> 5+2
>>> "viva"+"luis"
>>> 5*3
>>> "luis"*3
>>> "viva"*"luis"
>>> "luis"/3
>>> "luis"+3
>>> type(5)
>>> type(3.1416)
>>> type(4/2)
>>> 10/3
```

type is a *function* (more about this soon) that, within this context, gives the type of the variable

```
>>> 5+2
7
>>> "viva"+"luis"
'vivaluis'
>>> 5*3
15
>>> "luis"*3
'luisluisluis'
>>> "viva"*"luis"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
>>> "luis"/3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> "luis"+3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> type(5)
<class 'int'>
>>> type(3.1416)
<class 'float'>
>>> type(4/2)
<class 'float'>
>>> 10/3
3.3333333333333335
```

Note the final "5". There is a limit to the size and precision of the stored numbers.

Mathematical operators

Operator	Symbol	Action (numbers)	Action (strings)
Addition	+	Sum	Concatenate
Subtraction	-	Subtraction	none
Multiplication	*	Multiplication	Concatenate copies
Division	/	Division	none
Power	**	Power	none
Modulo	%	Remainder	none
Truncated division	//	Non-fractional part of result	none
Assignment	=	Assigns a number to variable	Assigns a string to variable

RETURN A STRING OR NUMBER

Comparative operators

Operator	Symbol
Equal to	==
Not equal to	!=, <>
Greater than	>
Less than	<
Greater or equal	>=
Less or equal	<=

Do not mistake the assignment operator (=) for the equality operator (==)

Logical operators

Operator	Symbol
And	and, & (bitwise)
Or	or, (bitwise)
Not	Not, !
Is	is
In	in

RETURN A BOOLEAN

Operators (comparative and logical)

Type the following statements in an interactive python interpreter and hit enter (shown just in the first one)

IMPORTANT: try to guess the result you will get before hitting enter)

```
>>> 5==2
>>> 5==5
>>> 5=5
>>> x=5
>>> x==5
>>> x==2
>>> y==5
>>> not(x==5)
>>> x>2 and x<100
>>> "a"<"b"
>>> x==5 or x==3
>>> "ACGT"=="ACGT"
>>> "ACGT"=="ACgT"
>>> "CG" in "ACGT"
>>> PassHPBBM==True
>>> PassHPBBM==1
```

```
>>> 5==2
False
>>> 5==5
True
>>> 5=5
File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> x=5
>>> x==5
True
>>> x==2
False
>>> y==5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> not(x==5)
False
>>> x>2 and x<100
True
>>> "a"<"b"
True
>>> x==5 or x==3
True
>>> "ACGT"=="ACGT"
True
>>> "ACGT"=="ACgT"
False
>>> "CG" in "ACGT"
True
>>> PassHPBBM==True
True
>>> PassHPBBM==1
True
```

Do not mistake the assignment operator (=) for the equality operator (==)

Use "&" only for bitwise comparisons!

For strings comparison is in lexicographic order

Case sensitive!

We defined PassHPBBM in the previous slide

Internally booleans are 0 (=False) or 1 (=True)

Functions can be thought of as little stand-alone programs that are called and executed within your program and return a specific calculation or carry out a specific task. Many function for common tasks come built into programming languages.

`function_name` ([parameter],[parameter],.....)

Functions (and methods) may expect zero, one, two,...or any number of values. These are called *parameters* or *arguments*.

Trailing set of parentheses is required even when the function takes no parameters

Examples of Built-in Functions

abs()
max()
help()
len()
min()
print()
sum()
type()

Examples of string type methods

str.upper()
str.lower()
str.count()
str.find()
str.replace()

Methods can be thought of as functions built into variables (see *objects*). They are used with the *dot notation*:

`variable_name.function_name` ([parameter],[parameter],.....)

String methods are extremely useful in bioinformatics to process nucleotide and protein sequences.

This are built-in functions but you can create your own functions as we'll see soon.

```
>>> abs()
>>> abs(-5)
>>> x=-5
>>> abs(x)
>>> y=abs(x)
>>> y
>>> MyDNA="ACGTGC"
>>> len(MyDNA)
>>> round(2.45687)
>>> round(2.95687)
>>> round(2.95687,2)
>>> z=input("type any number and then press enter")
>>> z
>>> type(z)
>>> int(z)
>>> help("round")
```

Help on built-in function *round*:

```
round(...)
round(number[, ndigits]) -> floating point number
```

Round a number to a given precision in decimal digits (default 0 digits).

This always returns a floating point number. Precision may be negative.

(END) *Press q (quick) to return to command shell*

```
>>> abs()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (0 given)
```

```
>>> abs(-5)
5
>>> x=-5
>>> abs(x)
5
>>> y=abs(x)
>>> y
5
```

Error, abs() function requires one argument to work and we didn't pass it

```
>>> MyDNA="ACGTGC"
>>> len(MyDNA)
6
>>> z=input("type any number and then press enter")
>>> z
```

(here whatever you typed)

```
>>> type(z)
<class 'str'>
```

Input() always returns a string!

```
>>> round(2.45687)
```

```
2
>>> help("round")
```

```
>>> round(2.95687,2)
2.96
```

Type the following statements in an interactive python interpreter and hit enter (shown just in the first one)

IMPORTANT: try to guess the result you will get before hitting enter)

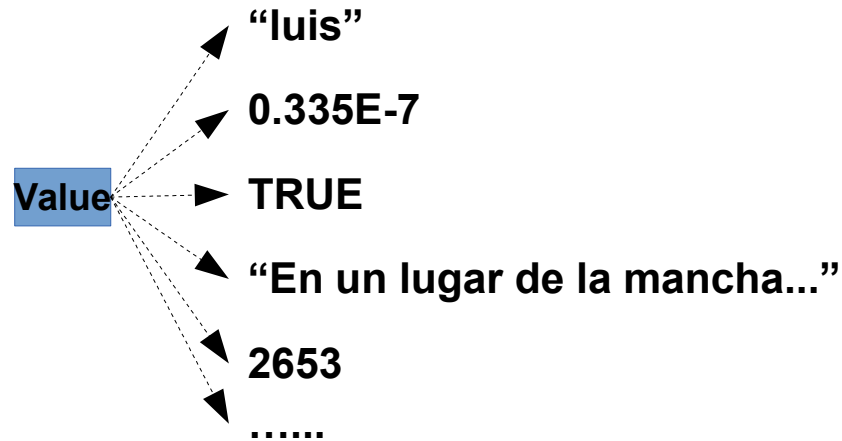
```
>>> MyDNA="AcagTaC"
>>> MyDNA
>>> MyDNA.upper()
>>> MyDNA
>>> MyDNA_Mayus=MyDNA.upper()
>>> MyDNA_Mayus
>>> MyDNA.lower()
>>> MyDNA.count("A")
>>> MyDNA_Mayus.count("A")
>>> MyDNA.upper().count("A")
>>> MyDNA.count("A").upper()
>>> MyDNA.find("a")
>>> MyDNA.find("a",3)
>>> MyDNA.replace("T","U")
>>> MyDNA.split("a")
>>> MyDNA_Mayus.split("a")
>>> MyDNA.split("a",1)
```


```
>>> MyDNA="AcagTaC"
>>> MyDNA
'AcagTaC'
>>> MyDNA.upper()
'ACAGTAC'
>>> MyDNA
'AcagTaC'
>>>
MyDNA_Mayus=MyDNA.upper()
>>> MyDNA_Mayus
'ACAGTAC'
>>> MyDNA.lower()
'acagtac'
>>> MyDNA.count("A")
1
>>> MyDNA_Mayus.count("A")
3
>>> MyDNA.upper().count("A")
3
>>> MyDNA.count("A").upper()
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'int' object has no
attribute 'upper'
>>> MyDNA.find("a")
2
>>> MyDNA.find("a",3)
5
```

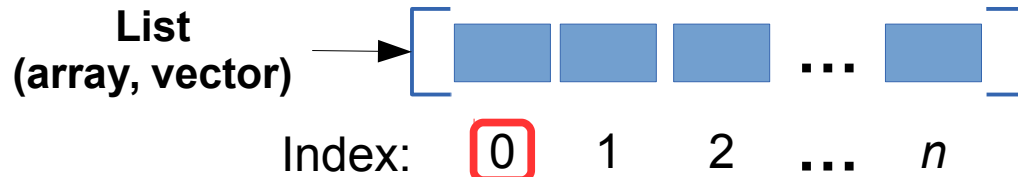
```
>>> MyDNA.replace("T","U")
'AcagUaC'
>>> MyDNA="ACGTGC"
>>> MyDNA.split("a")
['Ac', 'gT', 'C']
>>> MyDNA_Mayus.split("a")
['ACGTGC']
>>> MyDNA.split("a",1)
['Ac', 'gTaC']
```

Note that you can concatenate several methods in a single statement. However, beware of the order, they're evaluated from left to right.

By default, finds first occurrence



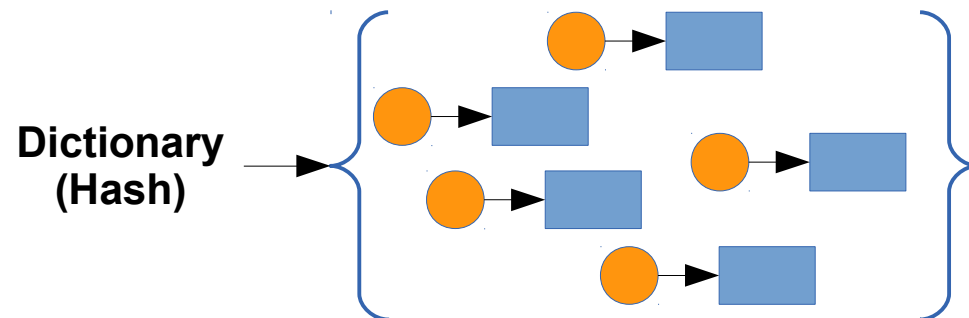
Simple variable → 



Sorted list of elements

Index: 0 1 2 ... n

Warning!!! first index is zero



"Bag" of pairs *key:element*

```
>>> Weight4Bq=[70,47,68,56,87,49,48,71,65,62]
>>> Weight4Bq
>>> Weight4Bq[0]
>>> Weight4Bq[3]
>>> Weight4Bq[-1]
>>> len(Weight4Bq)
```

We define (declare) a list by using square-brackets

We access list elements using square-brackets notation. Note that first element has index zero

How would you calculate the average weight in the class without typing weight values again but using the variable Weight4Bq?

```
>>> REenz={"EcoRI":"GAATTC", "BamHI":"GGATCC",
"HindIII":"AAGCTT", "NotI":"GCGGCCGC"}
>>> REenz["NotI"]
>>> REenz["EcoRI"]
>>> REenz.keys()
```

We define (declare) a dictionary by using curly-brackets and pairs key:value using colon

From the course's Moodle page, open the document InClassExercise_Dictionaries.txt and copy its content into the Python shell. What type of variable is HRAS? and REenz? How do you know it? Which of the enzymes above cut in the sequence of HRAS? (tip you just need to use the two variables in this exercise and one of the operators we've already seen)

```
>>> Weight4Bq=[70,47,68,56,87,49,48,71,65,62]
>>> Weight4Bq
[70, 47, 68, 56, 87, 49, 48, 71, 65, 62]
>>> Weight4Bq[0]
70
>>> Weight4Bq[3]
56
>>> Weight4Bq[-1]
62
>>> len(Weight4Bq)
10
>>> sum(Weight4Bq)
623
>>> sum(Weight4Bq)/len(Weight4Bq)
62
```

```
>>> REenz={"EcoRI":"GAATTC", "BamHI":"GGATCC",
"HindIII":"AAGCTT", "NotI":"GCGGCCGC"}
>>> REenz["NotI"]
'GCGGCCGC'
>>> REenz["EcoRI"]
'GAATTC'
>>> REenz.keys()
dict_keys(['BamHI', 'EcoRI', 'NotI', 'HindIII'])
>>> HRAS="GGTCCCGGC....."
>>> REenz["NotI"] in HRAS
True
>>> REenz["EcoRI"] in HRAS
False
```

- The nucleotide frequency in the genome of a newly discovered specie is given in the following dictionary: NucFrq={'A':0.35,'C':0.25,'G':0.3,'T':0.2}. Using the variable NucFrq, what would be the probability of finding the EcoRI restriction site ('GAATTC')? How many EcoRI sites would you expect if the genome is $2.7 \cdot 10^5$ bases long? Solve it using the interactive python shell and the provided variable (dictionary).

```
>>> NucFrq={'A':0.35,'C':0.25,'G':0.3,'T':0.2}
>>> (NucFrq['G']* (NucFrq['A']**2)* (NucFrq['T']**2)* NucFrq['C'])*270000
99.225
```

- The nucleotide frequency in the genome of a newly discovered specie is given in the following array: NucFrq=[0.4,0.3,0.2,0.1], where the values correspond to the frequencies of “A”, “C”, “G” and “T” respectively. Using the variable NucFrq, what would be the probability of finding the HindIII restriction site ('AAGCTT')? How many HindIII sites would you expect if the genome is $3.4 \cdot 10^9$ bases long? Solve it using the interactive python shell and the provided variable (array).

```
>>> NucFrq2=[0.4,0.3,0.2,0.1]
>>> (NucFrq2[1]* (NucFrq2[0]**2)* (NucFrq2[3]**2)* NucFrq2[2])*(3.4e9)
326400.0000000002
>>> round((NucFrq2[1]* (NucFrq2[0]**2)* (NucFrq2[3]**2)* NucFrq2[2])*(3.4e9))
326400
```

You are now ready to complete programming Assignment 1!

A) Python interactive interpreter

- 1) Open an interactive session with the python interpreter by typing “python3” in a terminal. Note that the prompt changes to “>>>” indicating we're now in a python shell.
- 2) Type the python commands to be executed (one statement at a time)
- 3) Exit the python shell by typing pressing ctrl key+D or typing:

>>> *exit()*

Note that all the code and results are now gone!

Terminal: text input/output environment through which you could interact with a computer

- On linux: Ctr+T
- On Mac OS: Command+space (search for “terminal”)
- On Windows: Start→Search “powershell”

B) IDLE (Integrated DeveLopment Environment for Python)

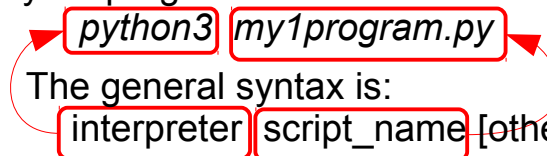
- 1) The main IDLE shell window is equivalent to the regular interactive interpreter with a few improvements
- 2) It includes a python code editor: go to “File-->New file” under the IDLE shell menu
- 3) The code editor allows you to save a list of commands and execute them in batch.
- 4) The code is saved as a plain text file: this is an **script**.

To open IDLE:

- On linux: type idle3 in a terminal
- On Windows/Mac: select IDLE from program menu

C) Running python from an script

- 1) Write program (code) in your favourite text editor.
- 2) Save it (remember it is a text file!). This is your source code (your script). For example save it as my1program.py
- 3) Run your program. To do this last step type in a terminal:

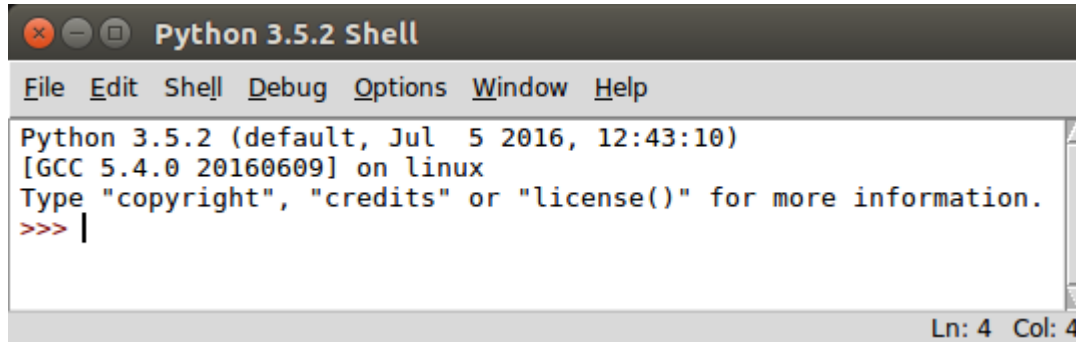

The general syntax is:
interpreter script_name [other arguments]

This send your source code (saved in the *my1program.py* text file) to the python interpreter (a program that translates your source code into machine code) who compiles it and pass the instructions to the CPU

D) More sophisticated IDEs (Integrated Development Environment)

Spyder, Ninja, SPE, ERIC, many others

1 Open IDLE

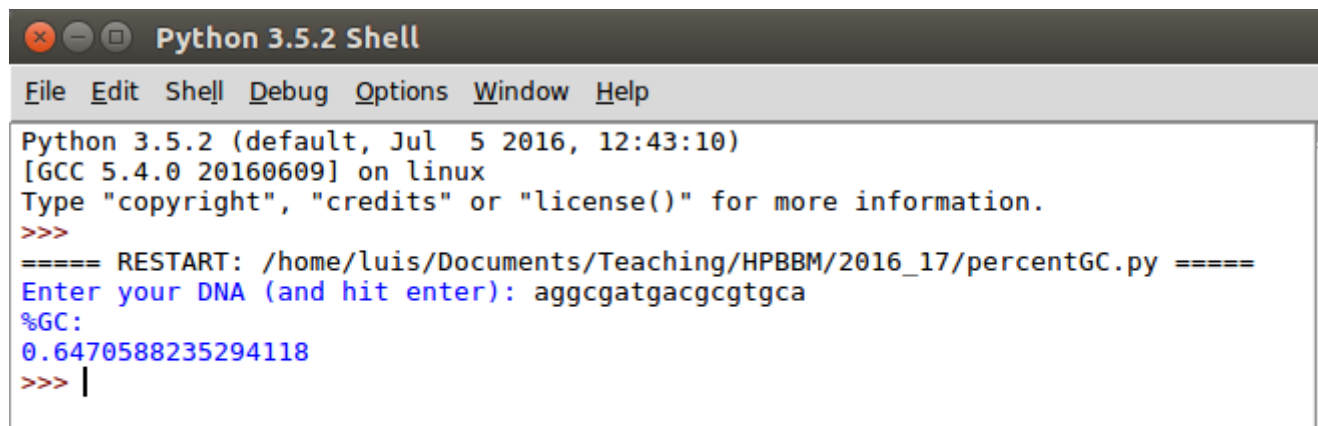
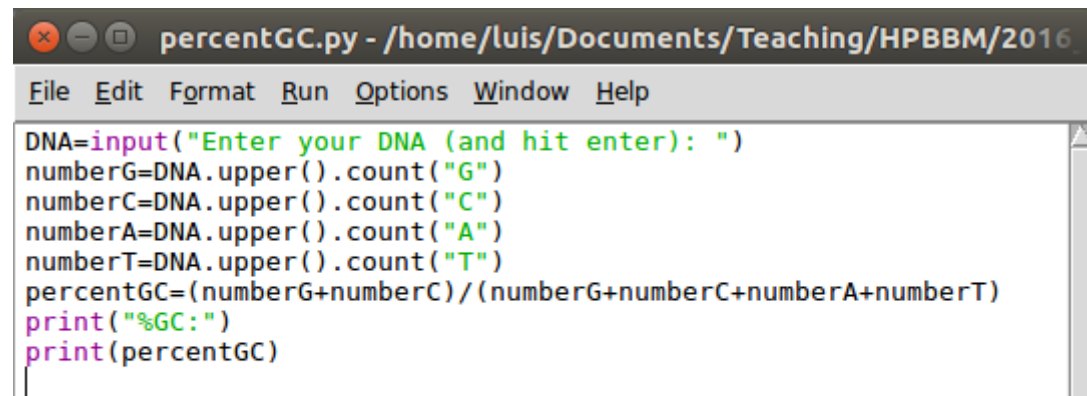


2 File→New File

3 Type commands

4 Save script:
File-->save

5 Run script:
Run-->Run module



You are now ready to
complete programming
Assignment 1!

Modifying lists and dictionaries

```
>>> MyList=list(range(2,20,3))
>>> MyList
>>> MyList[3]="x"
>>> MyList
>>> MyList=MyList+["a"]
>>> MyList
>>> MyList=["b","c"]+MyList
>>> MyList
>>> MyList.append("5")
>>> MyList
>>> MyList.insert(4,"new")
>>> MyList
>>> MyList.remove("x")
>>> MyList
>>> MyList.pop(4)
>>> MyList
>>> MyList.clear()
>>> MyList
>>> MyList=[34,5,14,3,1]
>>> MyList.reverse()
>>> MyList
>>> sorted(MyList)
>>> MyList
>>> MyList.sort()
>>> MyList
>>> MyList=[]
>>> MyList
```

```
>>> MyDic={'A':'Ala','G':'Gly','Y':'Tyr'}
>>> MyDic
>>> MyDic['F']='Phe'
>>> MyDic
>>> MyDic.keys()
>>> MyDic.values()
>>> MyDic.get('G')
>>> MyDic.get('S')
>>> MyDic.get('S','not found')
>>> del MyDic['G']
>>> MyDic
>>> del MyDic
>>> MyDic
>>> MyList1=['A','G','Y']
>>> MyList2=['Ala','Gly','Tyr']
>>> MyDic=dict(zip(MyList1,MyList2))
```



MyList = 

Index: 0 1 2 3 4

Index: -5 -4 -3 -2 -1

MyList[1:4] =  Elements [1,4)
that is 4 not included!

MyList[-4:-2] =  Elements [-4,-2)
that is -2 not included!

MyList[:4:2] =  Index may be omitted
Third value is the step

```
>>> a=[1,2,3,4,5,6,7,8,9,10]
>>> a
>>> a[2]
>>> a[-1]
>>> a[2:5]
>>> a[5:2]
>>> a[-5:-2]
>>> a[-2:-5]
>>> a[1:]
>>> a[1:8]
>>> a[1:8:2]
>>> a[::2]
>>> a[::3]
>>> a[:::-1]
```

```
>>> a=[1,2,3,4,5,6,7,8,9,10]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[2]
3
>>> a[-1]
10
>>> a[2:5]
[3, 4, 5]
>>> a[5:2]
[]
>>> a[-5:-2]
[6, 7, 8]
>>> a[-2:-5]
[]
>>> a[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[1:8]
[2, 3, 4, 5, 6, 7, 8]
>>> a[1:8:2]
[2, 4, 6, 8]
>>> a[::2]
[1, 3, 5, 7, 9]
>>> a[::3]
[1, 4, 7, 10]
>>> a[:::-1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In some respects strings behave *as if* they were lists of characters: *strings* can be *indexed* and *sliced*. The function *list()* and the method *join()* can be used to transform a string into a true list and a list into a string respectively.

MyString = “ **P** **y** **t** **h** **o** **n** ”

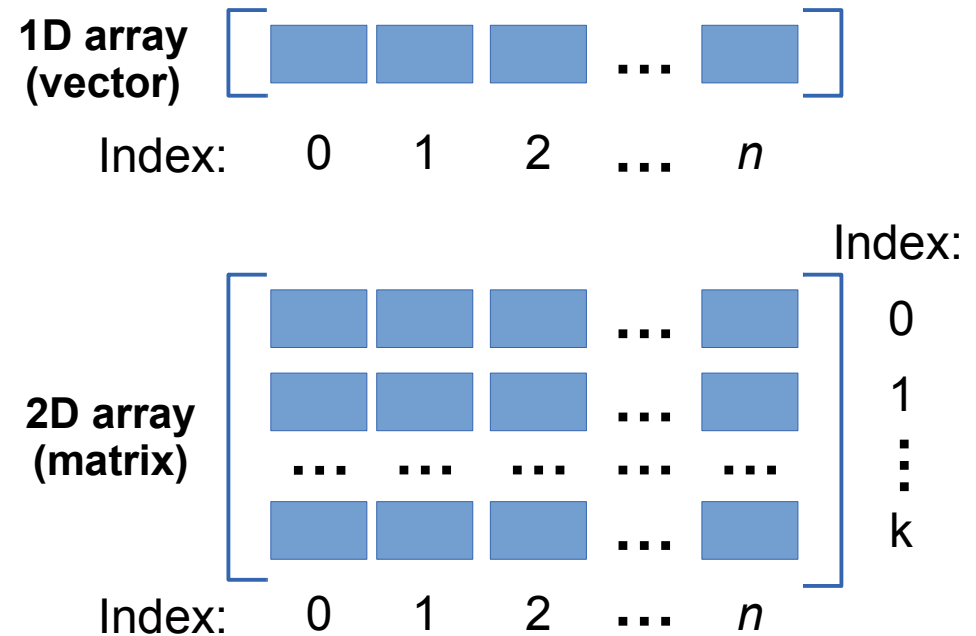
Index:	0	1	2	3	4	5
Index:	-6	-5	-4	-3	-2	-1

```
>>> MyDNA= 'ACGTGACGACCATGA'
>>> MyDNA[3]
>>> MyDNA[3:5]
>>> MyDNA[-1]
>>> MyDNA[3:]
>>> MyDNA[2:8:2]
>>> MyDNA[:: -1]
>>> list(MyDNA)
>>>
MyList= ["B", "i", "o", "c", "h", "e", "m", "i", "s", "t", "r", "y"]
>>> "-".join(MyList)
>>> "".join(MyList)
```

```
>>> MyDNA= 'ACGTGACGACCATGA'
>>> MyDNA[3]
'T'
>>> MyDNA[3:5]
'TG'
>>> MyDNA[-1]
'A'
>>> MyDNA[3:]
'TGACGACCATGA'
>>> MyDNA[2:8:2]
'GGC'
>>> MyDNA[:: -1]
'AGTACCAGCAGTGCA'
>>> list(MyDNA)
['A', 'C', 'G', 'T', 'G', 'A', 'C', 'G', 'A', 'C', 'C', 'A', 'T', 'G', 'A']
>>>
MyList= ["B", "i", "o", "c", "h", "e", "m", "i", "s", "t", "r", "y"]
>>> "-".join(MyList)
'B-i-o-c-h-e-m-i-s-t-r-y'
>>> "".join(MyList)
'Biochemistry'
```

So far we have studied one-dimensional arrays (also known as vectors). However, arrays can have any number of dimensions. In all the cases we use the square-bracket notation to access the array elements.

```
>>> a=[1,2,3,4,5,6]
>>> a
[1,2,3,4,5,6]
>>> a[2]
3
>>> aa=[[1,2,3],[4,5,6],[7,8,9]]
>>> aa
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> aa[1]
[4, 5, 6]
>>> aa[1][0]
4
>>> aa[2][2]
9
>>> aaa=[[[1,2],[3,4]],[[5,6],[7,8]]]
>>> aaa
[[[1,2],[3,4]],[[5,6],[7,8]]]
>>> aaa[1]
[[5, 6], [7, 8]]
>>> aaa[1][0]
[5, 6]
>>> aaa[1][0][1]
6
```



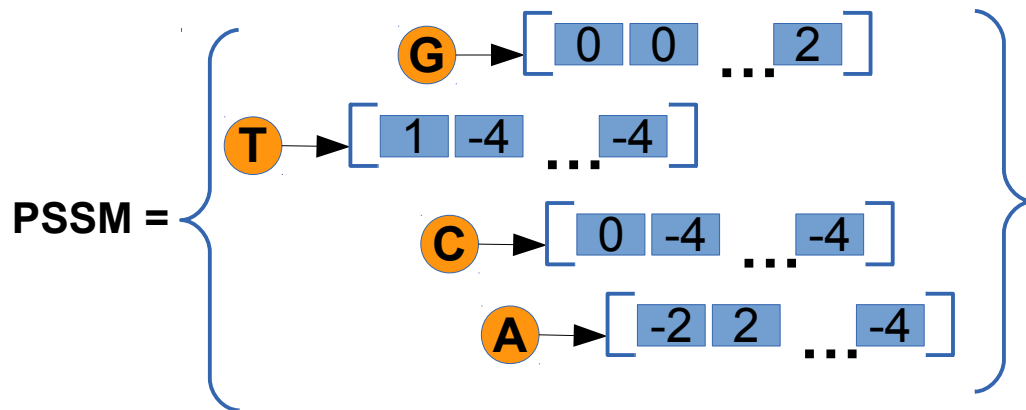
aa =

1	4	7
2	5	8
3	6	9

Not limited to 2D...

Similarly, dictionaries can also consists in more than one dimension. Moreover, we can combine arrays and dictionaries.

```
>>> PSSM={'A':[-2,2,-4,-4,-4,-4],'C':[0,-4,2,-4,-4,-4],'G':[0,0,-4,2,-4,2],  
-4,2],'T':[1,-4,-4,-4,2,-4]},}  
>>> PSSM  
{'G': [0, 0, -4, 2, -4, 2], 'T': [1, -4, -4, -4, 2, -4], 'C': [0, -4, 2, -4, -4, -4], 'A': [-2, 2, -4, -4, -4, -4]}  
>>> PSSM['A']  
[-2, 2, -4, -4, -4, -4]  
>>> PSSM['A'][3]  
-4
```



PSSM =

	1	2	...	n
A	-2	2	...	-4
C	0	-4	...	-4
G	0	0	...	2
T	1	-4	..	-4

Learning on-line:

- <http://software-carpentry.org/>
- <https://www.codecademy.com/learn>
- <https://www.edx.org/course/introduction-computer-science-mitx-6-00-1x-8>

On-line books:

- <https://learnpythonthehardway.org/book/>
- <http://greenteapress.com/thinkpython/html/index.html>

Programming aids

- <http://pythontutor.com/>
- <https://github.com/>
- <http://blog.coderscrowd.com/real-time-programming-for-bioinformatics-and-for-fun/>
- <http://scratch.mit.edu/>

Online Python Tutor: Embeddable Web-Based
Program Visualization for CS Education

Philip J. Guo
Google, Inc.
Mountain View, CA, USA

