

# Python for Bioinformatics Lecture 1

Lecturer: Pieter De Bleser  
Bioinformatics Core Facility, IRC



# Goals of this course

- Concepts of computer programming
- Rudimentary Python (widely-used language)
- Introduction to Bioinformatics file formats
- Practical data-handling algorithms
- Exposure to Bioinformatics software

# What is Python?

## Python is a Programming Language

Developed in the early 1990s by Guido van Rossum.

## Python Properties

1. Free
2. Interpreted Language
3. Object-Oriented
4. Cross-platform
5. Extensible, rich set of libraries
6. Popular for bioinformatics
7. Powerful
8. Widely used (Google, NASA, Yahoo, Electronic Arts, some Linux operating system scripts etc.)
9. Named after a British comedy “Monty Python’s Flying Circus”
10. Official: <http://www.python.org>
11. An overview of the web site:  
<https://www.python.org/about/gettingstarted/>



<https://gvanrossum.github.io/>

# How does it look like?

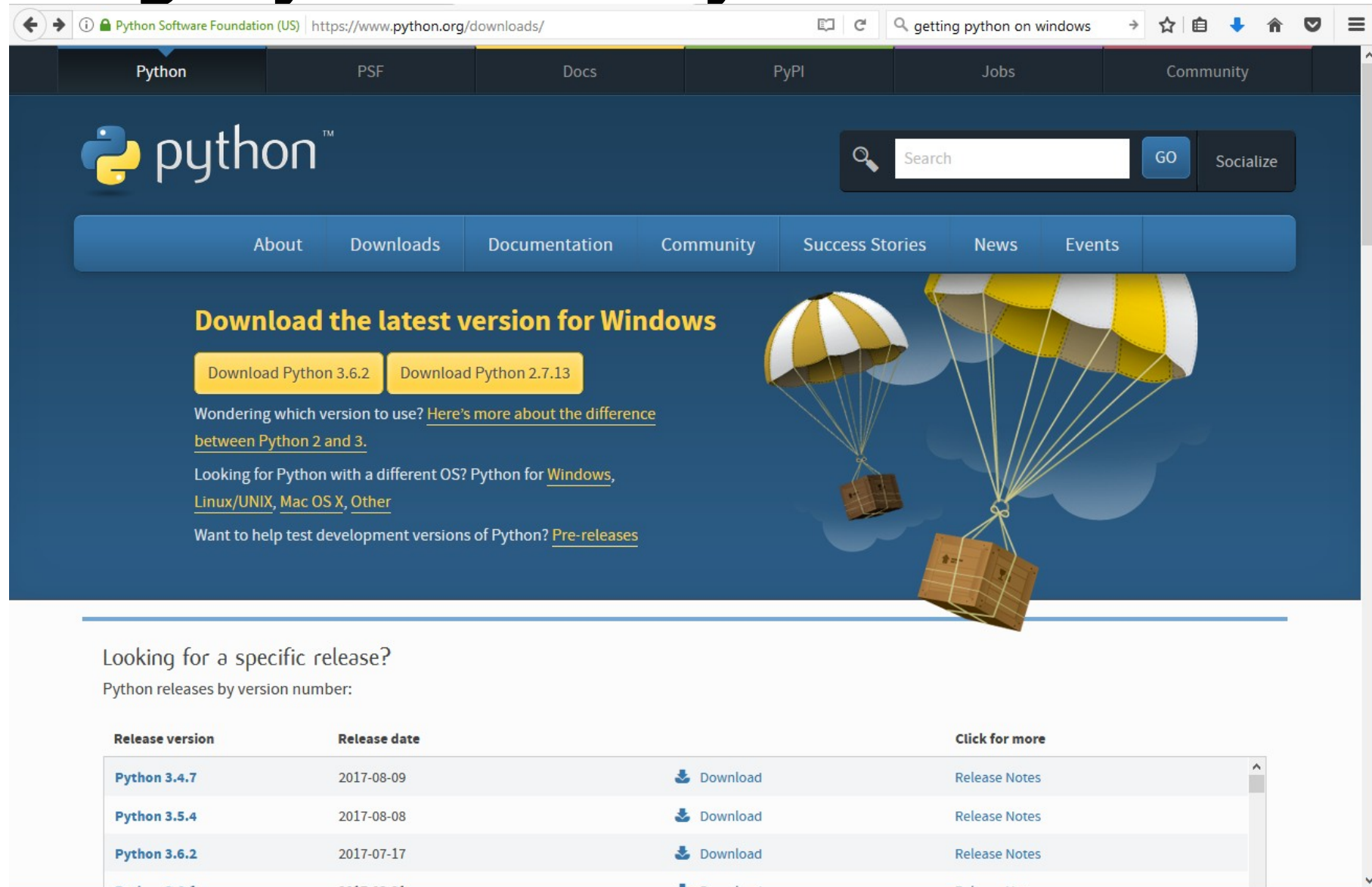
```
#!/usr/bin/env python3

#get the username from a prompt
username = input("Login: >> ")

#list of allowed users
user1 = "Jack"
user2 = "Jill"

#control that the user belongs to the
list of allowed users
if username == user1:
    print("Access granted")
elif username == user2:
    print("Welcome to the system")
else:
    print("Access denied")
```

# Getting Python for your OS



The screenshot shows the Python.org website's download page. The browser's address bar displays the URL <https://www.python.org/downloads/>. The page features a dark blue header with the Python logo and a navigation menu. Below the header, there's a search bar and a 'Socialize' button. The main content area has a large blue banner with the text 'Download the latest version for Windows' and two buttons for downloading Python 3.6.2 and Python 2.7.13. To the right of the buttons is an illustration of two parachutes carrying boxes. Below the banner, there's a section titled 'Looking for a specific release?' with a link to 'Here's more about the difference between Python 2 and 3.' and another link to 'Python for Windows, Linux/UNIX, Mac OS X, Other'. At the bottom, there's a table of Python releases by version number.

**Download the latest version for Windows**

[Download Python 3.6.2](#) [Download Python 2.7.13](#)

Wondering which version to use? [Here's more about the difference between Python 2 and 3.](#)

Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [Mac OS X](#), [Other](#)

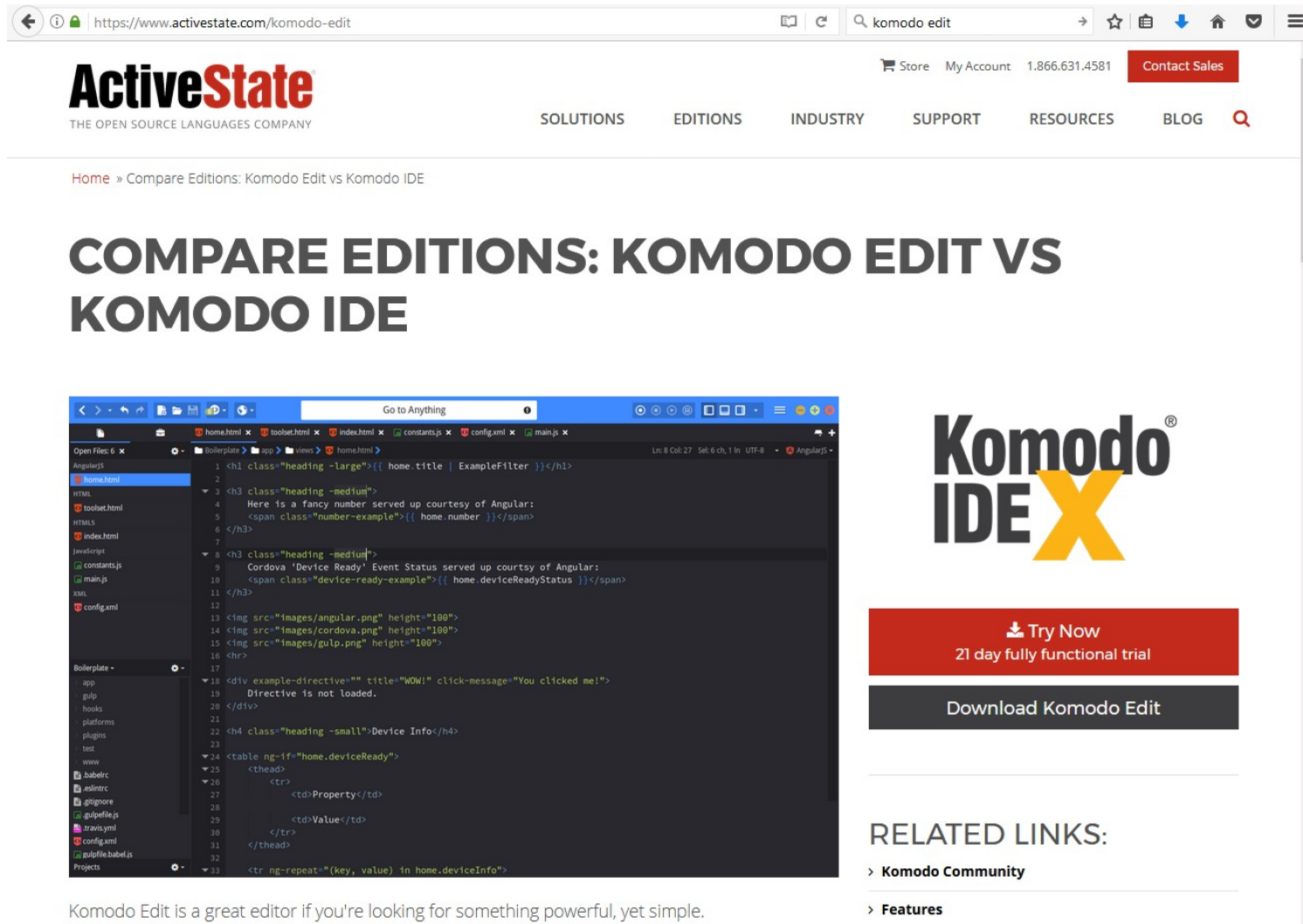
Want to help test development versions of Python? [Pre-releases](#)

Looking for a specific release?  
Python releases by version number:

Release version	Release date	Click for more
<a href="#">Python 3.4.7</a>	2017-08-09	<a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.5.4</a>	2017-08-08	<a href="#">Download</a> <a href="#">Release Notes</a>
<a href="#">Python 3.6.2</a>	2017-07-17	<a href="#">Download</a> <a href="#">Release Notes</a>

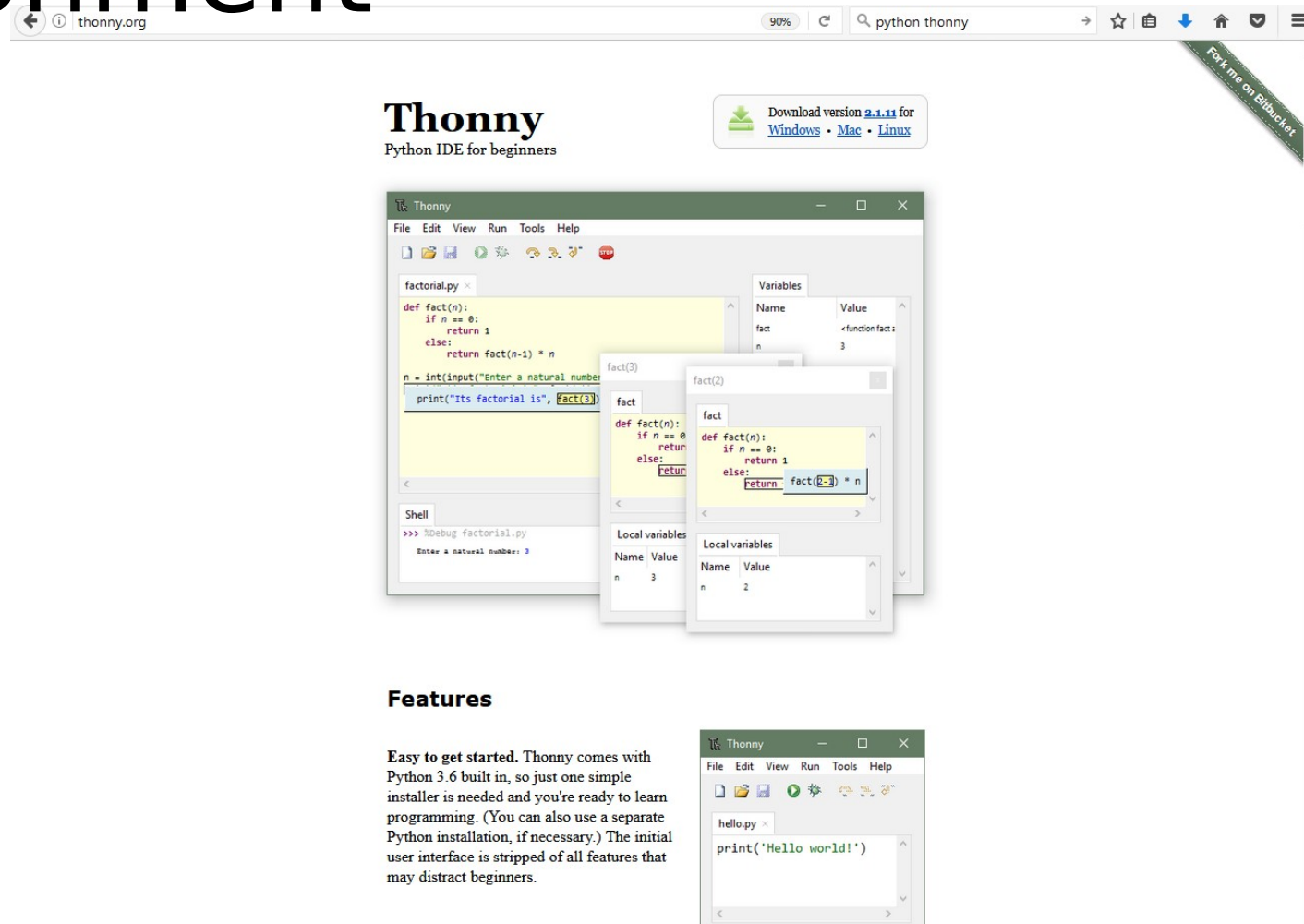
<https://www.python.org/downloads/>

# A Free Integrated Development Environment



<https://www.activestate.com/komodo-edit>

# A Free Integrated Development Environment



The image shows a screenshot of a web browser displaying the Thonny website at [thonny.org](http://thonny.org). The browser's address bar shows the URL, and the search bar contains "python thonny". The website header features the Thonny logo and the tagline "Python IDE for beginners". A download button for version 2.1.11 is visible, with links for Windows, Mac, and Linux. A green banner in the top right corner says "Fork me on GitHub".

Below the website header, a screenshot of the Thonny IDE interface is shown. The IDE window displays a Python script named `factorial.py` with the following code:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return fact(n-1) * n

n = int(input("Enter a natural number: "))
print("Its factorial is", fact(n))
```

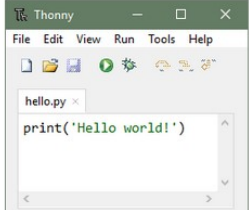
The IDE also shows a Shell window with the command `>>> %debug factorial.py` and the input `Enter a natural number: 3`. Overlaid on the IDE are three debug windows showing the state of the program during execution:

- fact(3) window:** Shows the function `fact` with `n=3`. The local variables table shows `n` with value `3`.
- fact(2) window:** Shows the function `fact` with `n=2`. The local variables table shows `n` with value `2`.
- fact(1) window:** Shows the function `fact` with `n=1`. The local variables table shows `n` with value `1`.

The **Variables** window shows the global variable `fact` as a function object and the local variable `n` with its current value.

## Features




**Easy to get started.** Thonny comes with Python 3.6 built in, so just one simple installer is needed and you're ready to learn programming. (You can also use a separate Python installation, if necessary.) The initial user interface is stripped of all features that may distract beginners.



A small screenshot of the Thonny IDE shows a file named `hello.py` with the code `print('Hello world!')`.

<http://thonny.org/>

<https://www.jetbrains.com/pycharm/download/>

[Tools](#)[Languages](#)[Solutions](#)[Support](#)[Company](#)[Store](#)

PyCharm

Coming in 2019.2


What's New

Features

Docs & Demos

Buy

Download



Version: 2019.1.3  
Build: 191.7479.30  
Released: May 30, 2019

[System requirements](#)  
[Installation Instructions](#)  
[Other versions](#)

## Download PyCharm

[Windows](#)[macOS](#)[Linux](#)

### Professional

For both Scientific and Web Python development. With HTML, JS, and SQL support.

DOWNLOAD


Free trial

### Community

For pure Python development

DOWNLOAD


Free, open-source



#### On Ubuntu?

PyCharm is now also available as a snap package. If you're on Ubuntu 16.04 or later, you can install PyCharm from the command line:

```
sudo snap install [pycharm-professional|pycharm-community] --classic
```



#### Using other JetBrains tools as well?

Get the [Toolbox App](#) to download PyCharm and its future updates with ease





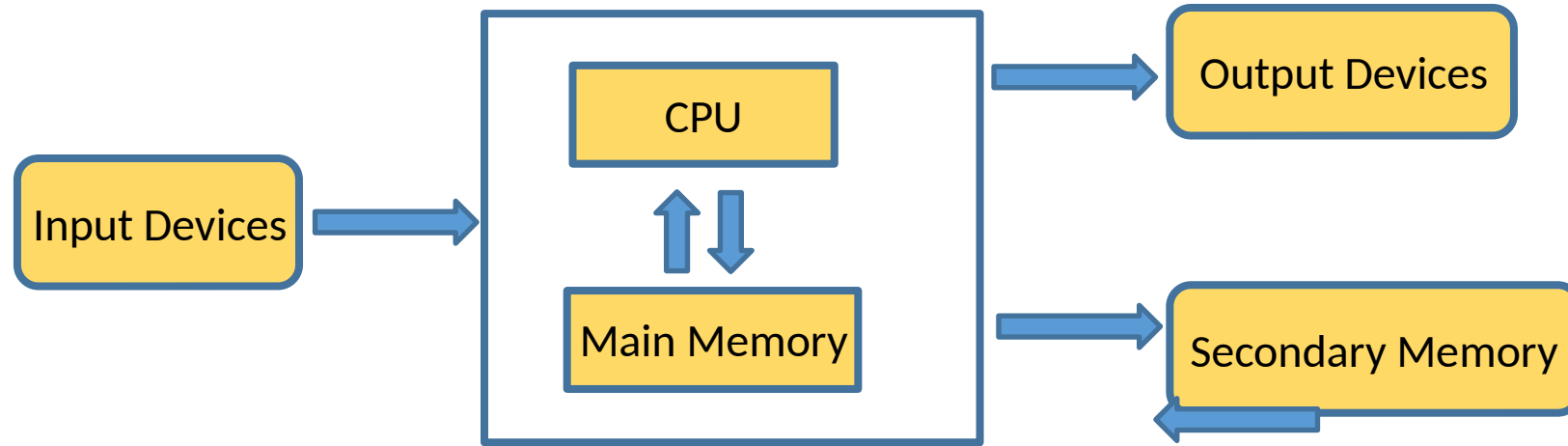
# Course setup

- Course web site:
  - <http://bioit2.irc.ugent.be/py4bio/>
- Github repository:
  - [https://github.ugent.be/pdbleser/PY4BIO\\_2019](https://github.ugent.be/pdbleser/PY4BIO_2019)
- For each lecture create a dedicated directory to save the program text files and your solutions to exercises
- For class teaching/demo I will use Anaconda/Spyder

# Setting the technological scene

Python's relationship to operating systems and applications

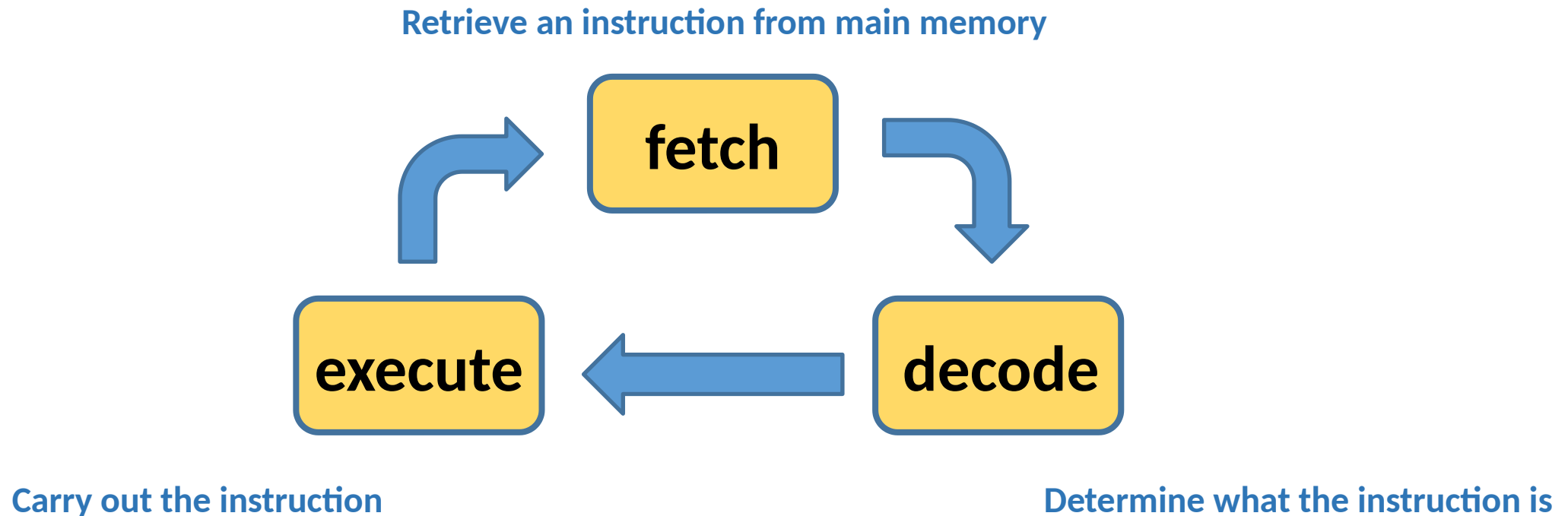
# Hardware Basics



- CPU (central processing unit) - the “brain” of the machine, where all the basic operations are carried out, such as adding two numbers or do logical operations
- Main Memory - stores programs & data. CPU can ONLY directly access info stored in the main memory, called RAM (Random Access Memory). Main memory is fast, but volatile.
- Secondary Memory - provides more permanent storage
  - Hard disk (magnetic)
  - Optical discs
  - Flash drives
- Input Devices - keyboard, mouse, etc.
- Output Device - monitor, printer, etc.

# How does the CPU executes a program?

- The instructions that comprise the program are copied from the secondary memory to the main memory
- CPU start executing the program, following a process called the: '**fetch execute cycle**'



# Programming Languages

- A program is simply a sequence of instructions telling a computer what to do.
- Programming languages are special notations for expressing computations in an exact, and unambiguous way
  - Every structure in a program language has a precise form (its **syntax**) and a precise meaning (its **semantics**)
- Python is one example of a programming language. Others include C++, Fortran, Java, Perl, Matlab, ...

# High-Level versus machine language

- Python, C++, Fortran, Java, and Perl are **high-level** computer languages, designed to be used and understood by humans.
- However, CPU can only understand very **low-level** language known as machine language

# High-Level versus machine language

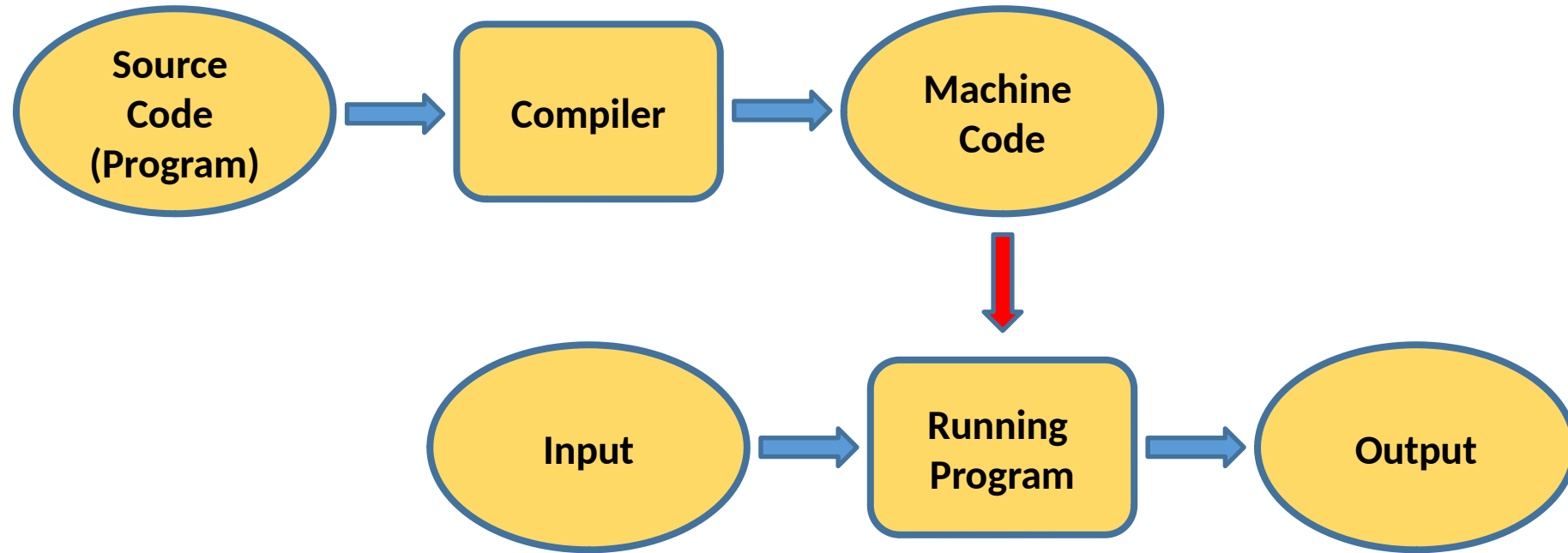
- The instructions that the CPU actually carries out might be something like this:
  1. Load the number from memory location 2001 into the CPU
  2. Load the number from memory location 2002 into the CPU
  3. Add the two numbers in the CPU
  4. Store the result into location 2003
- With instructions and numbers represented in binary notations (as sequences of 0s and 1s)
- In a high-level language (e.g. Python):  $c = a + b$



# Translate a high-level language to a machine language

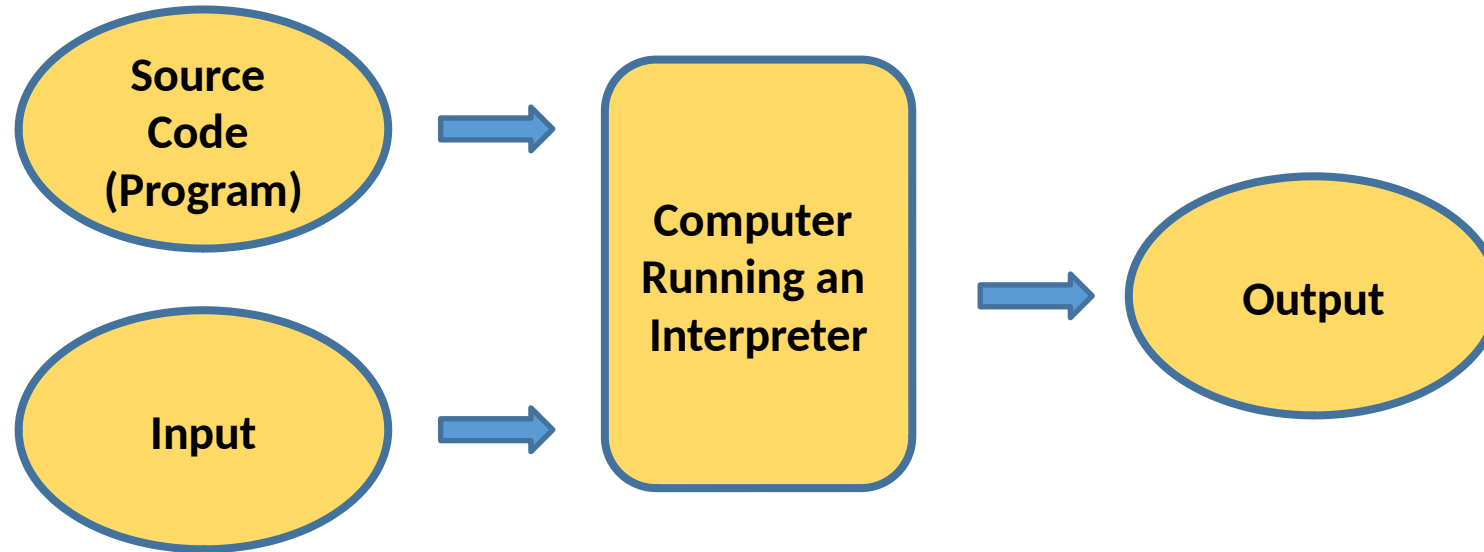
- Programs written in a high-level language need to be translated into the machine language the computer can execute
- Two ways to do this: a high-level language can be either **compiled** or **interpreted**

# Compiling a high-level language



- A **compiler** is a complex computer program that takes another program written in a high-level language and translates it into an equivalent program in the machine language of some computer

# Interpreting a high-level language



- An **interpreter** is a program that simulates a computer that understands a high-level language. Rather than translating the source program into a machine language equivalent, *the interpreter analyzes and executes the source code instruction by instruction* as necessary.
- To run a program, both the interpreter and the source are required each time.
- Interpreted languages tend to have more flexible programming environment as programs can be developed and run interactively, but are generally slower than compiled languages.

# Rules of Software Development

1. **Formulate Requirements** Figure out exactly what the problem to be solved is
2. **Determine Specifications** Describe exactly what your program will do.  
*What* will it accomplish? What the inputs and outputs of the program?
3. **Create a Design** Formulate the overall structure of the program.  
*How* will the program achieve the desired goals?
4. **Implement the Design** Translate the design into a computer language and put it into the computer.
5. **Test/Debug the Program** Try out your program and see if it works as expected. If there are any errors (often called *bugs*), then you should go back and fix them. The process of locating and fixing errors is called *debugging* a program.
6. **Maintain the Program** Continue developing the program in response to the needs of your users. Most programs are never really finished; they keep evolving over years of use.

# The Basics

Getting started with Python for Bioinformatics programming

# Let's Get Started!

Program name: welcome-1.py

File name: welcome-1.py

```
print( "Welcome to the Wonderful World of Bioinformatics!", end='\n' )  
print( "Welcome to the Wonderful World of Bioinformatics!", end=':' )  
print( "Welcome to the Wonderful World of Bioinformatics!" )
```

Command line output:

```
D:\PY4BIO\Lecture1>python welcome-1.py  
Welcome to the Wonderful World of Bioinformatics!  
Welcome to the Wonderful World of Bioinformatics!:Welcome to the Wonderful World of Bioinformatics!
```

# Creating/running programs (demo)

## Step 1: Writing your program

- Use a IDE (Integrated Development Environment) such as Komodo Edit allowing the creation and optional running of the programs or,
- Use a text editor (e.g. vi, Notepad, Notepad++) to enter the program.
  - Remember to **save it as a text file** ending with the suffix dot-py “.py”
  - In most cases, running the program will be done using the command line.

## Step 2: Translating and running your program

- You need to open a command line (DOS shell or Linux terminal) to translate/run your Python program.
  - The name of the Python translator is “python”.
  - To translate/run your program type “python welcome-1.py” at the command line.

# Spyder

The image shows the Spyder Python IDE interface. The top menu bar includes File, Edit, Search, Source, Run, Debug, Consoles, Projects, Tools, View, and Help. Below the menu is a toolbar with various icons for file operations and execution. The main window is divided into three panels:

- Editor:** The left panel shows a code editor with two tabs: 'hello.py' and 'welcome-1.py'. The 'welcome-1.py' tab is active, displaying the following Python code:

```
1 print( "Welcome to the Wonderful World of Bioinformatics!", end='\n' )
2 print( "Welcome to the Wonderful World of Bioinformatics!", end=':' )
3 print( "Welcome to the Wonderful World of Bioinformatics!" )
4
```

**Editor**

Type and save program text here...
- Help:** The right panel shows the 'print' function documentation. It includes the definition: `print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)` and the type: 'Function of builtins module'. It also provides a detailed description of the function's behavior and optional keyword arguments.

**Help**
- IPython console:** The bottom panel shows the IPython console output. It displays the results of running the code in the editor:

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type "copyright", "credits" or "license" for more information.

IPython 7.6.1 -- An enhanced Interactive Python.

In [1]: runfile('/home/pieterdb/ownCloud/PROG4BIO_2017/PY4BIO/Lecture1/welcome-1.py',
wdir='/home/pieterdb/ownCloud/PROG4BIO_2017/PY4BIO/Lecture1')
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!:Welcome to the Wonderful World of
Bioinformatics!

In [2]: runfile('/home/pieterdb/ownCloud/PROG4BIO_2017/PY4BIO/Lecture1/hello.py',
wdir='/home/pieterdb/ownCloud/PROG4BIO_2017/PY4BIO/Lecture1')
hello

In [3]:
```



**Python Interpreter**

Results will appear here...



# Another version of the 'welcome' script

Program name: welcome-2.py

File name: welcome-2.py

```
print( "Welcome ", end='' )  
print( "to ", end='' )  
print( "the ", end='' )  
print( "Wonderful ", end='' )  
print( "World ", end='' )  
print( "of ", end='' )  
print( "Bioinformatics!" )
```

Command line output:

```
D:\PY4BIO\Lecture1>python welcome-2.py  
Welcome to the Wonderful World of Bioinformatics!
```

# Snippet of Wisdom 1

Programs execute in sequential order

# Snippet of Wisdom 2

Less is better

# Snippet of Wisdom 3

If you can say something with fewer words, then do so

# Iteration

# Using the Python while construct

```
#!/usr/bin/env python

# The 'forever' program - a (Python) program,
# which does not stop until someone presses Ctrl-C.

import time

while ( True ):

    # Since Python 3.3 print() supports the keyword argument "flush".
    # Set to 'True' it disables output buffering.
    # Useful when running scripts within the IDE.

    print( "Welcome to the Wonderful World of Bioinformatics!", flush=True )
    time.sleep(1)
```

# Snippet of Wisdom 4

Add comments to make future maintenance of a program easier for other programmers and for you

# Running forever...(until you press Ctrl-C)

```
D:\PY4BIO\Lecture1>python forever.py
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Welcome to the Wonderful World of Bioinformatics!
Traceback (most recent call last):
  File "forever.py", line 14, in <module>
    time.sleep(1)
KeyboardInterrupt
```



# Running exactly ten times...

```
#!/usr/bin/env python

# The 'tentimes' program
# a (Python) program,
# which stops after ten iterations.

HOWMANY = 10
count = 0
while ( count < HOWMANY ):
    count = count + 1

    print( "Welcome to the Wonderful World of Bioinformatics!"
```

# Snippet of Wisdom 5

A condition can result in a value of true or false

# Introducing variable containers

Variables can represent any data type, not just integers:

```
my_string = 'Hello, World!'
my_flt = 45.06
my_bool = 5 > 9 #A Boolean value will return either True or False
my_list = ['item_1', 'item_2', 'item_3', 'item_4']
my_tuple = ('one', 'two', 'three')
my_dict = {'letter': 'g', 'number': 'seven', 'symbol': '&'}
```

# Snippet of Wisdom 6

When you need to change the value of an item, use  
a variable container

# Snippet of Wisdom 7

Don't be lazy: use good, descriptive names for variables

# Variable containers and loops

```
#!/usr/bin/env python

# The 'tentimes' program
# a (Python) program,
# which stops after ten iterations.

HOWMANY = 10
count = 0
while ( count < HOWMANY ):
    count = count + 1

    print( "Welcome to the Wonderful World of Bioinformatics!" )
```

# Running 'tentimes.py'

## Linux is a bit different:

```
$ chmod u+x tentimes.py # make tentimes.py executable
```

```
$ ./tentimes.py
```

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Welcome to the Wonderful World of Bioinformatics!

# Using the Python 'if' construct

```
#!/usr/bin/env python

# The 'fivetimes' program
# a (Python) program,
# which stops after five iterations.

HOWMANY = 5
count = 0
while ( True ):
    count = count + 1

    print( "Welcome to the Wonderful World of Bioinformatics!" )

    if ( count == HOWMANY ):
        break # break here
```



# There Really Is MTOWTDI (even with Python!)

```
#!/usr/bin/env python

# The 'oddeven' program - a (Python) program,
# which iterates four times, printing 'odd' when count
# is an odd number, and 'even' when count is an even
# number.

HOWMANY = 4

count = 0
while ( count < HOWMANY ):
    count = count + 1

    if ( count == 1 ):
        print( "odd" )
    elif ( count == 2 ):
        print( "even" )
    elif ( count == 3 ):
        print( "odd" )
    else: # at this point count is four.
        print( "even" )
```

# The oddeven2 program

```
#!/usr/bin/env python

# The 'oddeven-2' program - another version of 'oddeven'.

HOWMANY = 4

count = 0
while ( count < HOWMANY ):
    count = count + 1

    if ( count % 2 == 0 ):
        print( "even" )
    else: # count % 2 is not zero.
        print( "odd" )
```

# Using the modulus operator

```
print(5 % 2) # prints a '1' on a line.  
print(4 % 2) # prints a '0' on a line.  
print(7 % 4) # prints a '3' on a line.
```

# The oddeven-3 program

```
#!/usr/bin/env python
```

```
# The 'oddeven-3' program - another version of 'oddeven'.
```

```
HOWMANY = 4
```

```
count = 0
```

```
while ( count < HOWMANY ):
```

```
    count = count + 1
```

```
    even_or_odd = lambda: "even" if ( count % 2 == 0 ) else "odd"
```

```
    print( even_or_odd() )
```

# Snippet of Wisdom 8

There's more than one way to do it

# Processing Data Files

```
#!/usr/bin/env python

# The 'getlines' program which processes lines.

import sys

for line in sys.stdin:
    # What is the function of rstrip?
    # https://docs.python.org/3/library/stdtypes.html#str.rstrip
    print( line.rstrip() )
```

Running getlines.py:

```
D:\PY4BIO\Lecture1>python getlines.py < getlines.py
```

```
#!/usr/bin/env python
```

```
# The 'getlines' program which processes lines.
```

```
import sys
```

```
for line in sys.stdin:
    # What is the function of rstrip? https://docs.python.org/3/library/stdtypes.html#str.rstrip
    print( line.rstrip() )
```

# Running getlines...

Make getlines executable on Linux machines:

```
$ chmod u+x getlines.py
```

```
$ ./getlines.py
```

# Use getlines to view at the content of files

```
$ ./getlines.py < patterns.py
```



# Introducing patterns

```
#!/usr/bin/env python
```

```
# The 'patterns' program which searches for patterns in lines of text.
```

```
import fileinput
```

```
import re
```

```
for line in fileinput.input():
```

```
    if re.search( "even", line ):
```

```
        print( line.rstrip() )
```

# Running patterns

```
pieterdb@DESKTOP-LB3OVT0 MINGW32 /d/PROG4BIO_2017/PY4BIO/Lecture1
$ python patterns.py oddeven.py
# The 'oddeven' program - a (Python) program,
# is an odd number, and 'even' when count is an even
    print( "even" )
    print( "even" )
```

# Program format

- No end-of-line character (no semicolons!)
- **Whitespace** matters (4 spaces for indentation)
- No extra code needed to start (no "public static ...")
- For clarity, it is recommended to write each **statement** in a separate line, and use indentation in nested structures.
- **Comments**: Anything from the # sign to the end of the line is a comment.
- A python **script** consists of all of the Python statements and comments of the file taken collectively as one big routine to execute.

# Python's style guides

Python's style guide:

<http://www.python.org/dev/peps/pep-0008/>

Google's Python style guide:

<http://google-styleguide.googlecode.com/svn/trunk/pyguide.html>

Other:

pychecker: <http://pychecker.sourceforge.net/>

pyflakes: <https://launchpad.net/pyflakes/>

# A minimal Python program

Lines  
beginning  
with "#" are  
comments,  
and are ignored  
by Python

```
#!/usr/bin/env python3  
  
#Elementary Python program  
print('Hello World!')
```

The 'shebang' line (optional)

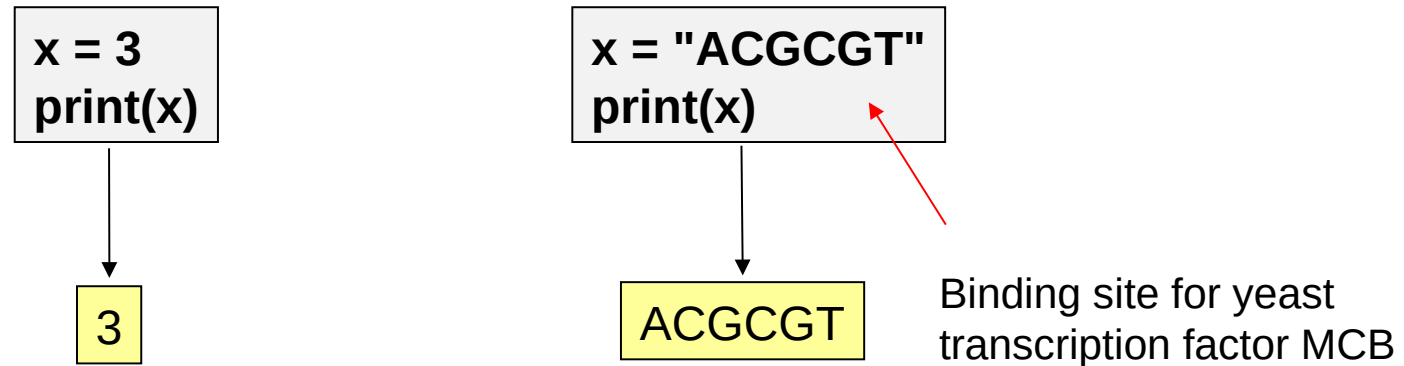
Single or double quotes  
enclose a "string literal"

**print** statement tells Python to print the following stuff to the screen

```
Hello World
```

# Variables

- ✓ We can tell Python to "remember" a particular value, using the assignment operator "=":



- ✓ The `x` is referred to as a "scalar variable".
- ✓ *Variable names can contain:*
  - ✓ *alphabetic characters,*
  - ✓ *numbers (but not at the start of the name), and*
  - ✓ *underscore symbols `_`*

# Variables and Objects

Everything in Python is an object

- An object models a real-world entity
- Objects possess methods (also called functions):
  - Methods are typically applied to the object, possibly parameterized
- Objects can also possess variables, that describe their state
  - e.g. `x.upper()` is a parameter-less method, that works on the string object `x`

Object . Method or variable

# Built-in datatypes and operations

- **Truth Values and Boolean Operations:**
  - `None`, `True`, `False` (are special type of integers)
  - Boolean Operations: `and`, `or`, `not`
- **Comparisons:**
  - `<`, `==`, `!=`, `is not`, ...
- **Numeric types:**
  - Integers and floating point numbers
  - Arithmetic operations: `x+y`, `x/y`, `abs(x)`, `x**y`, ...
- **Sequence types:**
  - Strings, lists, tuples, ...
  - Operations: concatenation (`+`), `len(x)`, `x in s`, ...
  - Strings with additional methods:
    - `capitalize`, `endswith`, `find`, `islower`, `lstrip`, ...
- **Set types:**
  - `set`, `frozenset` (can't be changed: immutable)
  - Operations: `len(s)`, `issubset(other)`, `union(s1, s2)`, `add`, ...
- **Mapping Types:**
  - `dict`: maps hashable keys to values
  - Operations: `del d[key]`, `key in d`, ...
- ... and some more



# Arithmetic operations

- Basic operators are + - / \* %

```
x = 14
y = 3
print("Sum: ", x + y)
print("Product: ", x * y)
print("Remainder: ", x % y)
```

Sum: 17  
Product: 42  
Remainder: 2

Could write  
x \*= 2



Could write  
x += 1



```
x = 5
print("x started as ", x)
x = x * 2
print("Then x was ", x)
x = x + 1
print("Finally x was ", x)
```

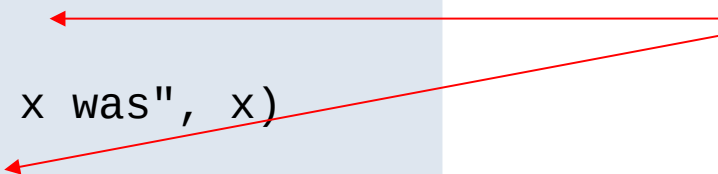
x started as 5  
Then x was 10  
Finally x was 11

# Or interactively...

```
>>> x = 14
>>> y = 3
>>> x + y
17
>>> x * y
42
>>> x % y
2
>>> x = 5
>>> print("x started as", x)
x started as 5
>>> x *= 2
>>> print("Then x was", x)
Then x was 10
>>> x += 1
>>> print("Finally x was", x)
Finally x was 11
>>>
```

- This way, you can use Python as a calculator
- Can also use:

`+=` `-=` `/=` `*=`



# String operations

- Concatenation

+ +=

```
a = "pan"  
b = "cake"  
a = a + b  
print(a)
```



pancake

```
a = "soap"  
b = "dish"  
a += b  
print(a)
```



soapdish

- Can find the length of a string using the function len(x)

```
mcb = "ACGCGT"  
print("Length of %s is "%mcb, len(mcb))
```



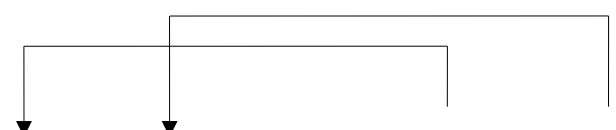
Length of ACGCGT is 6

# String formatting

Strings can be formatted with place holders for inserted strings (%s) and numbers (%d for digits and %f for floats)

- Use Operator % on strings:

Formatted string	%	Insertion tuple
------------------	---	-----------------



```
>>> "aaaa%saaa%saaa"%("gcgcg", "tttt")
'aaaagcgcgcaaaatttttaa'
>>> "A range written like this: (%d - %d)" % (2,5)
'A range written like this: (2 - 5)'
>>> "Or with preceeding 0's: (%03d - %04d)" % (2,5)
"Or with preceeding 0's: (002 - 0005)"
>>> "Rounding floats %.3f" % math.pi
'Rounding floats 3.142'
>>> "Scientific notation: %.3e" % 0.0000002345)
'Scientific notation: 2.345e-07'
```

# Print formatting I

```
print("My name is {}".format("Luka"))  
myName="Khan"  
print("My name is {}".format(myName))
```



```
> python3 print_format-1.py  
My name is Luka.  
My name is Khan.
```

- A data type is a '**class**'; Python provides default functions that you can do **from** such a class
  - These are called *methods*
  - The object (class) determines which verbs (methods) you can use

# Print formatting II

```
# left align
print("Yes, {:10s} is my name.".format("Luka"))
print("Yes, {:<10s} is my name.".format("Luka"))

myName="Khan"

# left align
print("Yes, {:<10s} is my name.".format(myName))
# center align
print("Yes, {:^10s} is my name.".format(myName))
# right align
print("Yes, {:>10s} is my name.".format(myName))
```

You can also determine how much space the formatting code will take in the output  
Try varying the 10 and see what happens...



```
> python3 print_format-2.py
Yes, Luka          is my name.
Yes, Luka          is my name.
Yes, Khan          is my name.
Yes,      Khan     is my name.
Yes,              Khan is my name.
```

### Formatting Options

**Format Specification** ::= `[[fill]align][sign][#][0][width][.][.precision][type]`

**fill** ::= `<any character>`

**align** ::= `"<" | ">" | "=" | "^"`  
# Forces the field to be left(<), right(>), centered(^) aligned.

**sign** ::= `"+" | "-" | ""`  
# + indicates that a sign should be used for both positive as well as negative numbers.  
# - only for Negative number.

**width** ::= integer  
**precision** ::= integer

**type** ::= `"b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"`  
# The type determines how the data should be presented. The available integer presentation types are listed above: ONLY 's' is a String format

**PYN**  
pynative.com

# Print formatting Integers

```
print("This is {}".format(25))
print("This is {} and {}".format(25, 30))

print("This is %d.%25)
print("This is %i and %d"%(25, 30))
```



```
> python3 print_format-3.py
This is 25.
This is 25 and 30.
This is 25.
This is 25 and 30.
```

## Difference between %i and %d?

Python3 str.format() specification has dropped the support for "i" (%i or {:i}).

It only uses "d" (%d or {:d}) for specifying integers.

Therefore, you can simply use {:d} for all integers.

For output, i.e. for printf or logging, %i and %d are actually same thing, both in Python and in C.

There is a difference but only when you use them to parse input, like with scanf().

For scanf, %d and %i actually both mean signed integer but %i interprets the input as a hexadecimal number if preceded by 0x and octal if preceded by 0 and otherwise interprets the input as decimal.

Therefore, for normal use, it is always better to use %d, unless you want to specify input as hexadecimal or octal.

# Print formatting - Special Characters

```
print("The \\sign \n can \t also \t be \t printed.")  
  
print("He said: \"Hello\".")  
#print("He said: \"Hello\".")  
print('He said: "Hello".')
```



```
> python3 print_format-4.py  
The \sign  
  can    also    be      printed.  
He said: "Hello".  
He said: "Hello".
```



# Print formatting - floats

```
myFloat= 4545.4542244  
print("Print the full float: {},\ncutoff to 2 decimals: {:.2f}, \nor large with 1 decimal:  {:10.1f}.".format(myFloat, myFloat, myFloat))
```



```
> python3 print_format-5.py  
Print the full float: 4545.4542244,  
cutoff to 2 decimals: 4545.45,  
or large with 1 decimal:      4545.5.
```

# More string operations

Convert to upper case

Convert to lower case

Reverse the string

Translate "i"s into "a"s

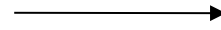
Calculate the length of the string

```
x = "A simple sentence"
print(x)
print(x.upper())
print(x.lower())
xl = list(x)
xl.reverse()
print("".join(xl))
x = x.replace("i","a")
print(x)
print(len(x))
```

```
A simple sentence
A SIMPLE SENTENCE
a simple sentence
ecnetnes elpmis A
A sample sentence
17
```

# Concatenating DNA fragments

```
dna1 = "accacgt"  
dna2 = "taggtct"  
print(dna1 + dna2)
```



accacgtaggtct

## Transcribing DNA to RNA

```
dna = "accACgttAGGTct"  
rna = dna.lower().replace("t", "u")  
print(rna)
```

DNA string is a mixture  
of upper & lower case

Make it all  
lower case

Turn "t" into "u"



accacguuaggucu

# Searching in strings

```
rna = "accacguuaggucu"  
pattern = "cguu"  
print(rna.find(pattern))
```



4

accacguuaggucu  
012345678...

```
rna = "accacguuaggucu"  
result = rna.endswith("gg")  
print(result)
```



False

```
rna = "accacguuaggucu"  
print("cguu" in rna)
```



True

# Conditional blocks

- ✓ The ability to execute an *action* contingent on some *condition* is what distinguishes a computer from a calculator. In Python, this looks like this:

Consistent, level-wise  
indenting important

```
if condition:  
    ↔ action  
else:  
    ↔ alternative
```

These indentations  
tell Python which  
piece of code  
is contingent on  
the condition.

```
x = 149  
y = 100  
if (x > y):  
    print(x, " is greater than ", y)  
else:  
    print(x, "is less than ", y)
```

```
149 is greater than 100
```

# Conditional operators

Numeric: > >= < <= != ==

"does not equal"

Note that the test for "x equals y" is `x==y`, **not** `x=y`

```
x = 5 * 4  
y = 17 + 3  
if x == y: print(x, "equals", y)
```

20 equals 20

## The same operators work on strings as alphabetic comparisons

Shorthand syntax for assigning more than one variable at a time

```
(x, y) = ("Apple", "Banana")  
if y > x: print(y, "after", x)
```

Banana after Apple

# Logical operators

- ✓ Logical operators: and and or

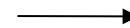
```
x = 222
if x % 2 == 0 and x % 3 == 0:
    print(x, "is an even multiple of 3")
```



222 is an even multiple of 3

- ✓ The keyword not is used to negate what follows. Thus not  $x < y$  means the same as  $x \geq y$
- ✓ The keyword False (or the value zero) is used to represent falsehood, while True (or any non-zero value, e.g. 1) represents truth. Thus:

```
if True: print("True is true")
if False: print("False is true")
if -99: print("-99 is true")
```



True is true  
-99 is true

# Exercise

Write a boolean expression that evaluates to:

True if the variable response starts with the letter "q", case-insensitive,  
False if it does not.



# Loops

✓ Here's how to print out the numbers 1 to 10:

The indented code is repeatedly executed as long as the condition  $x < 10$  remains true

```
x = 0
while x < 10:
    print(x, end=" ")
    x+=1
```

Equivalent to  
 $x = x + 1$

1 2 3 4 5 6 7 8 9 10

✓ This is a *while loop*.

The code is executed *while* the condition is true.

# A common kind of loop

- ✓ Let's dissect the code of the while loop again:

Initialisation →

Test for completion →

Continuation →

```
x = 0
while x < 10:
    print(x, end=" ")
    x+=1
```

- ✓ This form of while loop is common enough to have its own shorthand: the *for loop*.

Iteration variable

Generates a list

```
for x in range(10):
    print(x, end=" ")
```

# For loop features

- Loops can be used with all iterable types, ie.: lists, strings, tuples, iterators, sets, file handlers

```
>>> for nucleotide in "actgc":  
...     print(nucleotide,end=" ")  
...  
a c t g c
```

- Stepsizes can be specified with the third argument of the slice constructor (negative values for iterating backwards)

```
>>> for number in range(0,50,7) :  
...     print(number,end=" ")  
...  
0 7 14 21 28 35 42 49  
>>> for nucleotide in "actgc"[::-1]:  
...     print(nucleotide, end=" ")  
...  
c g t c a  
>>> print("HtaEdsLfgLdfOf"[::3])  
HELLO
```

# Enumerate

- Enumerate is a handy method to track the index when looping over a sequence
- Instead of ...

```
>>> i = 0
>>> for nuc in "actgc":
...     print("%i: %s" % (i+1, nuc))
...     i += 1
...
1: a
2: c
3: t
4: g
5: c
```


- We can let do enumerate do the work for us...

```
>>> for i, nuc in enumerate("actgc"):
...     print("%i: %s" % (i+1, nuc))
...
1: a
2: c
3: t
4: g
5: c
```

# Reading data from files

- To read from a file, we can conveniently iterate through it linewise with a for-loop and the open function.
- Internally a filehandle is maintained during the loop.

```
>>> for line in open("sequence.txt"):
...     print(line.rstrip())
...
>NC_000020
GGCCATGGTCAGCGTGAACG
CGCCCCTCGGGGCTCCAGTG
GAGAGTTCTTACGGTAAGTG
```

 rstrip() removes the trailing newline that is present at the end of each line in the file. What happens if we omit rstrip() in the code?

This code snippet opens a file called "sequence.txt" in the current directory, and iterates through it line by line

# Exercises

- 1)sum up all the numbers in `[3,7,10,4,-1,0]` ?
- 2)print all word in `['Oranges', 'Bananas', 'Cucumbers', 'Apples']`  
starting with an O or A ?
- 3)check which element of `[1,5,-1,8,7]` are also inside `[8,7,10,5,0,0]` ?
- 4)check if the ratio of two integers a and b is larger than 0.5?
- 5)check for all tuples in `[(3,2),(10,5),(1,-1)]` if the square of the first  
number can be divided by the second without rest?

# Summary

- Everything in Python is an object
- Built-in datatypes: numeric, sequences, sets, mappings
- Control structures: Loops and Conditions
  - For loop & while loop
  - if, else, elif, <, >, ==, !=, etc.
- Python well suited for string manipulation
  - Lots of string-specific methods