Introduction to Python language

What is Python?

> a modern, general-purpose, object-oriented, high-level programming language.

General characteristics

- **clean and simple language:**
  - Easy-to-read and intuitive code
  - easy-to-learn minimalistic syntax
  - maintainability scales well with size of projects
- **expressive language:**
  - Fewer lines of code
  - fewer bugs
  - easier to maintain

Python: high level technical details

- **dynamically typed:**
  - No need to define the type of variables, function arguments or return types.
- **automatic memory management:**

    - No need to explicitly allocate and deallocate memory for variables and data arrays: *no memory leak bugs*
- **interpreted:**
    - No need to compile the code: *the Python interpreter reads and executes the python code directly*

Python for scientific computing

Python has a strong position in *scientific computing*: large community of users, easy to find help and documentation

Extensive ecosystem of *scientific libraries* and environments

- **numpy** http://numpy.scipy.org (http://numpy.scipy.org) - Numerical Python
- **scipy** http://www.scipy.org (http://www.scipy.org) - Scientific Python
- **pandas** http://www.pydata.org (http://www.pydata.org) - Data analysis
- **matplotlib** http://www.matplotlib.org (http://www.matplotlib.org) - Plotting library
- **seaborn** http://seaborn.pydata.org/ (http://seaborn.pydata.org/) - Statistical data visualization

```
``To understand the meaning of the numbers we compute, we often
 need postprocessing, statistical analysis and graphical visuali
zation of our data.``
```

Python interpreter

The standard way to use the Python programming language is to use the Python interpreter to run Python code

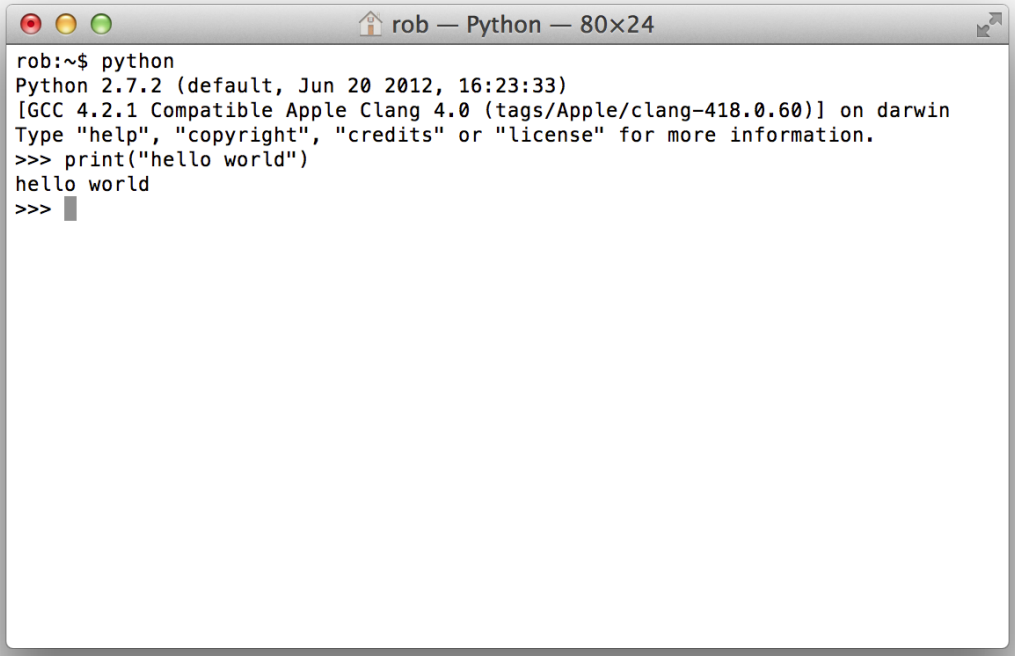
- The python interpreter is a program that reads and executes the python code in files passed to it as arguments
- At the command prompt, the command `python` is used to invoke the Python interpreter

For example, to run a file `my-program.py` that contains python code from the command prompt, use:

```
$ python my-program.py
```

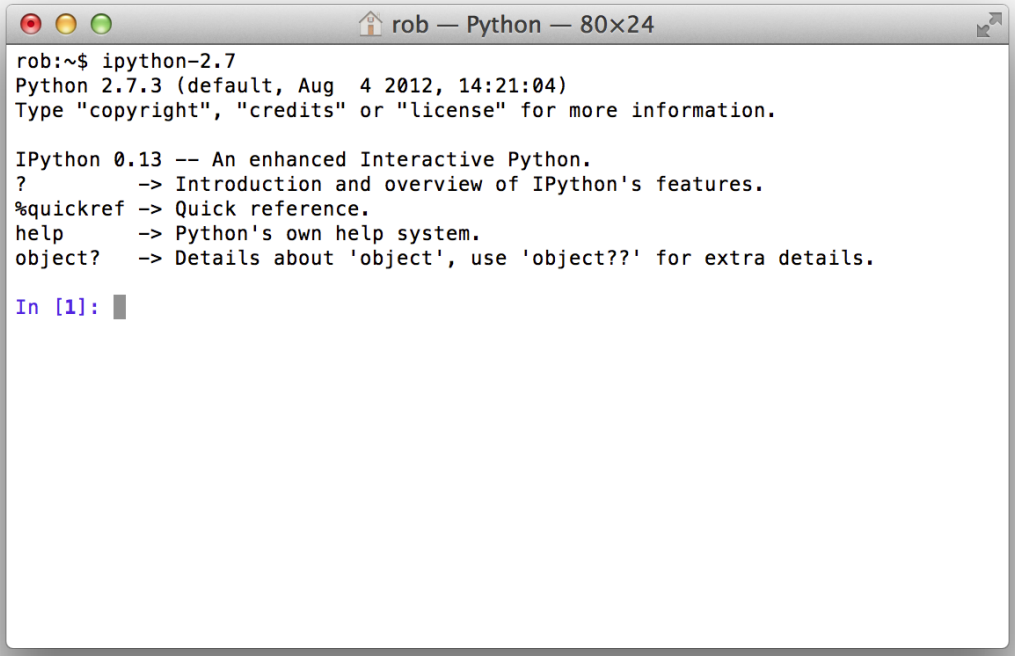
Python interactive shell

We can also start the interpreter by simply typing `python` at the command line, and interactively type python code into the interpreter.



IPython (1)

- IPython is an interactive shell that addresses the limitation of the standard python interpreter: it is a work-horse for scientific use of python!
- It provides an interactive prompt to the python interpreter with a greatly improved user-friendliness.



IPython (2)

Some of the many useful features of IPython includes:

- Command history, which can be browsed with the up and down arrows on the keyboard.
- Tab auto-completion.
- In-line editing of code.
- Object introspection, and automatic extract of documentation strings from python objects like classes and functions.
- Good interaction with operating system shell.

Using Python as a Calculator (1)

A Python interactive shell could be used as a powerful calculator

```
In [1]: 2+5
```

```
Out[1]: 7
```

```
In [3]: (50-5*6)/4
```

```
Out[3]: 5.0
```

In [4]:
```
7/3
```

Out[4]:
```
2.3333333333333335
```

Note:

- in Python 3 the integer division returns a floating point number;
- in Python 2, like in C or Fortran, the integer division truncates the remainder and returns an integer.

Using Python as a Calculator (2)

Our *Python Calculator* supports matematical functions, simply importing the `math` library

In [5]:
```
# An example of using a module
from math import sqrt
sqrt(81)
```

Out[5]:
```
9.0
```

In [6]:
```
# Or you can simply import the math library itself
import math
math.sqrt(81)
```

Out[6]:
```
9.0
```

Using Python as a Calculator (3)

In our *Python calculator*, we can define variables using the equal sign (=):

In [7]:
```
width = 20
length = 30
area = length*width
area
```

Out[7]:
```
600
```

Using Python as a Calculator (4)

If you try to access a variable that you haven't yet defined, you get an error:

In [8]:
```
volume
```

```
---------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-8-0c7fc58f9268> in <module>()
----> 1 volume

NameError: name 'volume' is not defined
```

and you need to define it:

In [9]:
```
depth = 10
volume = area*depth
volume
```

Out[9]:
```
6000
```

Python variables (1)

- You can name a variable *almost* anything you want
- It needs to start with an alphabetical character or "_"
- It can contain alphanumeric characters plus underscores ("_")

Certain words, however, are **reserved** for the *language*:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Python variables (2)

In [10]:
```
# Trying to define a variable using
# one of these will result in a syntax error:
return = 0
```

```
  File "<ipython-input-10-0cf476473b74>", line 3
    return = 0
           ^
SyntaxError: invalid syntax
```

A short introduction to Python programming language

(Python 3)

Strings (1)

Strings are lists of printable characters, and can be defined using either single quotes

In [11]:
```
'Hello, World!'
```

Out[11]:
```
'Hello, World!'
```

or double quotes

In [12]:
```
"Hello, World!"
```

Out[12]:
```
'Hello, World!'
```

Strings (2)

Single quotes and double quotes cannot be used both at the same time, unless you want one of the symbols to be part of the string.

In [13]:
```
"He's a Rebel"
```

Out[13]:
```
"He's a Rebel"
```

In [14]:
```
'She asked, "How are you today?"'
```

Out[14]:
```
'She asked, "How are you today?"'
```

In [15]:
```
myString = "I'm a string"
type(myString)
```

Out[15]:
```
str
```

Just like the other two data objects we're familiar with (ints and floats), you can assign a string to a variable

In [16]:
```
greeting = "Hello, World!"
```

How to concatenate strings (1)

You can use the + operator to concatenate strings together:

In [20]:
```
statement = "Hello," + "World!"
print(statement)
```

```
Hello,World!
```

Don't forget the space between the strings, if you want one there.

In [21]:
```
statement = "Hello, " + "World!"
print(statement)
```

```
Hello, World!
```

How to concatenate strings (2)

You can use + to concatenate multiple strings in a single statement:

In [22]:
```
print("This " + "is " + "a " + "longer " + "statement.")
```

```
This is a longer statement.
```

If you have a lot of words to concatenate together, there are other, more efficient ways to do this. But this is fine for linking a few strings together.

Lists

Very often in a programming language, one wants to keep a group of similar items together.

Python does this using a data type called **list**.

In [23]:
```
days_of_the_week = ["Sunday","Monday","Tuesday",\
                    "Wednesday","Thursday","Friday","Saturday"]
```

In [24]:
```
type(days_of_the_week)
```

Out[24]:
```
list
```

How to access to list items

You can access members of the list using the **index** of that item:

In [25]:
```
days_of_the_week[2]
```

Out[25]:
```
'Tuesday'
```

Python lists, like C, but unlike Fortran, use 0 as the index of the first element of a list.

In [26]:
```
# First element
print(days_of_the_week[0])

# If you need to access the *n*th element from the end of the list,
# you can use a negative index.
print(days_of_the_week[-1])
```

```
Sunday
Saturday
```

Some operations on lists

- You can add an additional item to the list using the .append() method:

In [27]:
```
languages = ["Fortran","C","C++"]
languages.append("Python")
print(languages)
```

```
['Fortran', 'C', 'C++', 'Python']
```

- You can concatenate two or more lists using the + operator

In [28]:
```
other_languages = ["Java", "Perl"]
my_languages = languages + other_languages
print(my_languages)
```

```
['Fortran', 'C', 'C++', 'Python', 'Java', 'Perl']
```

Lists: data type of items

A list **DOES NOT** have to hold the *same data type*. For example,

In [29]:
```
my_multitype_list = ["Today",7,99.3,""]
print(my_multitype_list,type(my_multitype_list))
```

```
['Today', 7, 99.3, ''] <class 'list'>
```

However, it's good (but not essential) to use a list for similar objects that are somehow logically connected.

The `range` data structure

- `range(stop)`: return an object that produces a sequence of integers from 0 to stop (exclusive)
- `range(start,stop[, step])`: return an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step; the default value for step is 1!
- The builtin function `list()` is useful for generating a list from a range object

In [30]:
```
list(range(10))
```

Out[30]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Other numerical lists

In [31]:
```
list(range(2,8))
```

Out[31]:
```
[2, 3, 4, 5, 6, 7]
```

In [32]:
```
evens = list(range(0,20,2))
evens
```

Out[32]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [33]:
```
evens[3]
```

Out[33]:
```
6
```

Number of elements in a sequence

In [34]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

In [35]:
```
#You can find out how long a list is using the **len()** command:
len(evens)
```

Out[35]:
```
10
```

Iteration, Indentation, and Blocks

One of the most useful things you can do with lists is to *iterate* through them

> i.e. to go through each element one at a time

To do this in Python, we use the **for** statement:

In [36]:
```
# Define loop
for day in days_of_the_week:
    # This is inside the block :)
    print(day)
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Blocks?

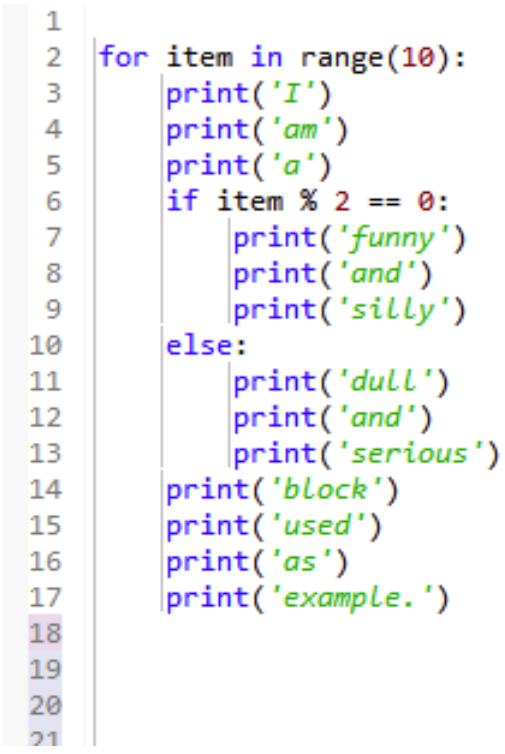
(Almost) every programming language defines blocks of code in some way.

- In Fortran, one uses DO .. ENDDO (or IF .. ENDIF, etc..) statements to open and close a code block.
- In C, C++, and Perl, one uses curly brackets {} to define blocks.

Python blocks

Python uses a colon (":"), followed by indentation level

> Everything at a the same level of indentation is taken to be in the same block.



```
for item in range(10):
    print('I')
    print('am')
    print('a')
    if item % 2 == 0:
        print('funny')
        print('and')
        print('silly')
    else:
        print('dull')
        print('and')
        print('serious')
    print('block')
    print('used')
    print('as')
    print('example.')
```

The `range()` class and `for` statement

The **range()** class is particularly useful with the **for** statement to execute loops of a specified length:

In [37]:
```
for i in range(12):
    print ("The square of",i,"is",i*i)
```

```
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
The square of 11 is 121
```

Slicing (1)

Warning: pay attention! Slicing is very important for using matrices and ***numpy***

Lists and strings have something in common that you might not suspect: they can both be treated as sequences.

You can iterate through the letters in a string:

In [39]:
```
for letter in "Sunday":
    print(letter)
```

```
S
u
n
d
a
y
```

Slicing (2)

More useful is the *slicing* operation on any sequence.

In [40]:
```
days_of_the_week[0:2]
```

Out[40]:
```
['Sunday', 'Monday']
```

or simply

In [41]:
```
days_of_the_week[:2]
```

Out[41]:
```
['Sunday', 'Monday']
```

Note: we are not talking about *indexing* anymore.

Slicing (3)

If we want the last items of the list, we can do this with negative slicing:

In [42]:
```
days_of_the_week[-3:]
```

Out[42]:
```
['Thursday', 'Friday', 'Saturday']
```

We can extract a subset of the sequence:

In [43]:
```
workdays = days_of_the_week[1:6]
print(workdays)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Slicing (4)

Since strings are sequences

In [44]:
```
day = "Sunday"
abbreviation = day[:3]
print(abbreviation)
```

```
Sun
```

Slicing (5)

We can pass a *third* element into the slice.

It specifies a step length (like the third argument of the **`range()`** class)

In [45]:
```
numbers = list(range(0,40))
evens = numbers[2::2]
evens
```

Out[45]:
```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 3
8]
```

note: I was even able to omit the second argument

Fundamental types (1)

The basic types in any language are:

- Strings (we already saw them)
- Integers
- Real
- Boolean

In [47]:
```
# integers
x = 1
type(x)
```

Out[47]:
```
int
```

In [48]:
```
# float
x = 1.0
type(x)
```

Out[48]:
```
float
```

Fundamental types (2)

In [49]:
```
# boolean
b1 = True

type(b1)
```

Out[49]:
```
bool
```

In [50]:
```
# complex numbers: note the use of `j` to specify the imaginary part
x = 1.0 - 2.0j
type(x)
```

Out[50]:
```
complex
```

In [51]:
```
print(x)
```

```
(1-2j)
```

In [52]:
```
print(x.real, x.imag)
```

```
1.0 -2.0
```

Checking type of a variable

In [53]:
```
x = 1.0

# check if the variable x is a float
type(x) is float
```

Out[53]:
```
True
```

In [54]:
```
# check if the variable x is an int
type(x) is int
```

Out[54]:
```
False
```

Type casting (1)

In [55]:
```
x = 1.5

print(x, type(x))
```

```
1.5 <class 'float'>
```

In [56]:
```
x = int(x)

print(x, type(x))
```

```
1 <class 'int'>
```

In [57]:
```
z = complex(x)

print(z, type(z))
```

```
(1+0j) <class 'complex'>
```

Type casting (2)

Some conversions are impossible:

In [58]:
```
x = float(z)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-e719cc7b3e96> in <module>()
----> 1 x = float(z)

TypeError: can't convert complex to float
```

Booleans and Truth Testing

- A **boolean** variable can be either *True* or *False*

- We invariably need some concept of *conditions* in programming
  - to control the branching behavior
  - to allow a program to react differently to different situations

`if` statement (1)

`if` statement controls the branching on the basis of a boolean value

In [60]:
```
if day == "Sunday":
    print("Sleep in")
else:
    print("Go to work")
```

```
Sleep in
```

Let's take the snippet apart to see what happened.

In [61]:
```
# First, note the statement
day == "Sunday"
```

Out[61]:
```
True
```

`if` statement (2)

If statements can have **elif** parts ("else if"), in addition to if/else parts. For example:

In [62]:
```
if day == "Sunday":
    print("Sleep in")
elif day == "Saturday":
    print("Do sport")
else:
    print("Go to work")
```

```
Sleep in
```

Equality testing

The == operator performs *equality testing*: if the two items are equal, it returns `True`, otherwise it returns `False`.

You can compare any data types in Python:

In [63]:
```
1 == 2
```

Out[63]:
```
False
```

In [64]:
```
50 == 2*25
```

Out[64]: True

In [65]:
```
3 < 3.14159
```

Out[65]: True

Other tests

In [66]:
```
1 != 0
```

Out[66]: True

In [67]:
```
2 <= 1
```

Out[67]: False

In [68]:
```
2 > 1
```

Out[68]: True

In [69]:
```
1 == 1.0
```

Out[69]: True

a "strange" equality test

Particularly interesting is the **`1 == 1.0`** test

hint: the two objects are different in terms of *data types* (integer and floating point number) but they have the same *value*

In [70]:
```
# A strange test
print(1 == 1.0)

# Operator **is** tests whether two objects are the same object
print(1 is 1.0)
```

```
True
False
```

More on boolean tests

We can do boolean tests on lists as well:

In [71]:
```
[1,2,3] == [1,2,4]
```

Out[71]: False

In [72]:
```
[1,2,3] < [1,2,4]
```

Out[72]:
```
True
```

Finally, note that you can also perform multiple comparisons in a single line; the result is a very intuitive test!

In [73]:
```
hours = 5
0 < hours < 24
```

Out[73]:
```
True
```

Dictionaries (1)

- **Dictionaries** are an object called "mappings" or "associative arrays" in other languages.
- Whereas a list associates an integer index with a set of objects:

```
mylist = [1,2,9,21]
```

- In a dictionary, the index is called the `key`, and the corresponding dictionary entry is the `value`

In [90]:
```
ages = {"Rick": 46, "Bob": 86, "Fred": 21}
print("Rick's age is",ages["Rick"])
ages
```

```
Rick's age is 46
```

Out[90]:
```
{'Bob': 86, 'Fred': 21, 'Rick': 46}
```

Dictionaries (2)

There's also a convenient way to create dictionaries without having to quote the keys.

In [91]:
```
dict(Rick=46,Bob=86,Fred=20)
```

Out[91]:
```
{'Bob': 86, 'Fred': 20, 'Rick': 46}
```

In [92]:
```
## looping on a dictionary
for key,value in ages.items():
    print(key,"is",value,"years old")
```

```
Fred is 21 years old
Rick is 46 years old
Bob is 86 years old
```

About dictionaries

- dictionaries are the most powerful structure in python

- dictionaries are *not* suitable for *everything*

In [93]:
```
len(t)
```

Out[93]: 4

In [94]:
```
len(ages)
```

Out[94]: 3