Computational Physics Laboratory

Python scripting

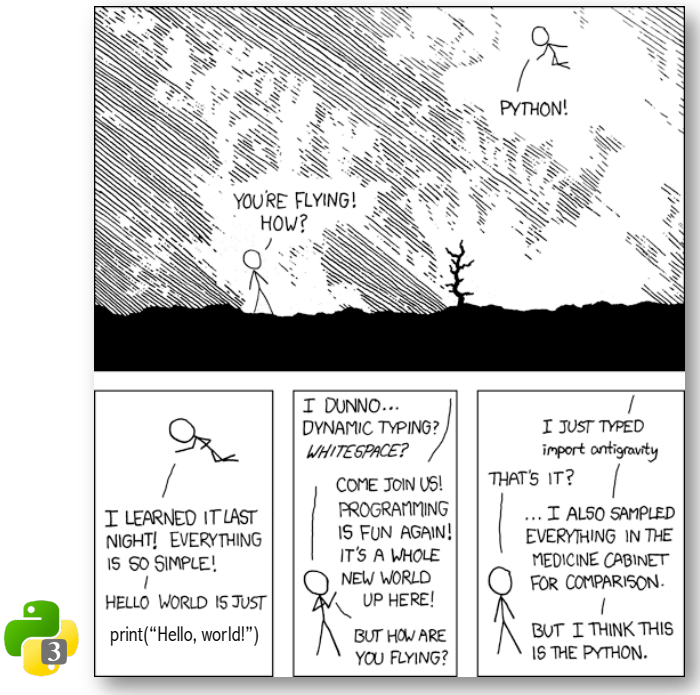
LECTURE 1

BASIC CONCEPTS

CONTROL STRUCURES

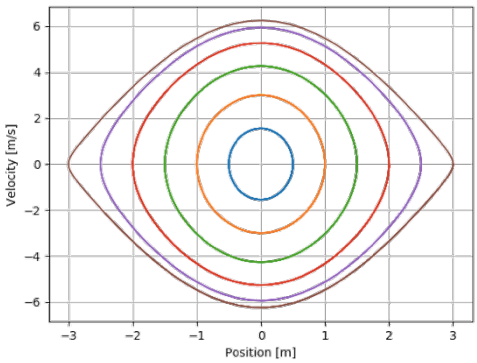
Outline

- What is Python ?
- Running Python
- Simple operations
- Strings, integers, floats & type conversions
- Input/Output
- Variables
- Booleans
- Comparison
- If, else & elif statements
- While



Example: Ordinary Differential Equation numerical integrator with symbolic calculus and plot of trajectories in the phase space

```
from sympy import *
from sympy.abc import x
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import ode   # Standing on the shoulders of giants!
m = 1
g = 9.8
L = 1
V = m*g*L*(1-cos(x)) # potential energy
F = -diff(V,x)       # force via symbolic differentiation!
a = F/m              # acceleration
def f(t,position):   # function: [x,v]->[v,a]
    return [position[1], a.evalf(subs={x: position[0]})]
solver = ode(f).set_integrator('dopri5') # 4th order Runge-Kutta solver
instants = 200
t = np.linspace(0.01, 7, instants)
plt.figure(1)
plt.grid()
plt.xlabel('Position [m]')
plt.ylabel('Velocity [m/s]')
for j in range(6):
    start = [(j+1)/2, 0.] # initial value
    solver.set_initial_value(start)
    r = np.zeros(instants)
    v = np.zeros(instants)
    for i in range(instants):
        r[i] = solver.integrate(t[i])[0]
        v[i] = solver.integrate(t[i])[1]
    plt.plot(r,v)
plt.show()
```



What is Python ?

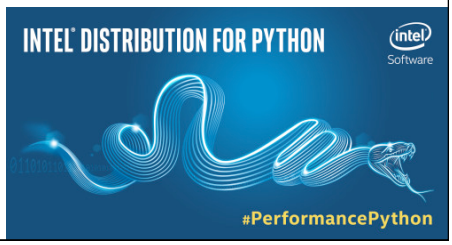
- Python is a high-level programming language, with applications in numerous areas, including scientific computing, machine learning, artificial intelligence, scripting , quantum computing, and web programming.
- It is very popular and used by organizations such as Google, NASA, etc.
- Python is processed at runtime by the interpreter, which is is a program that runs scripts. There is no need to compile your program before executing it.

What is Python ?

- Python is an easy to learn. It has efficient high-level data structures and a simple but effective approach to object-oriented programming.
- Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.
- Python is also suitable as an extension language for customizable applications.

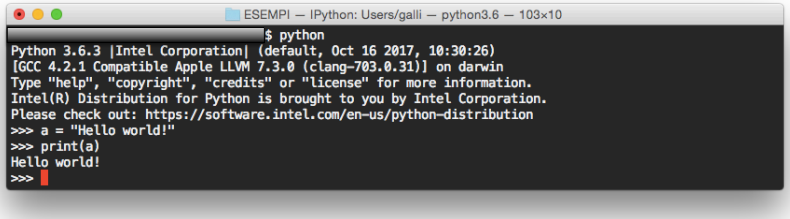
Welcome to Python!

- The three major versions of Python are 1.*, 2.* and 3.*. These are subdivided into minor versions, such as 2.7 and 3.6.
- Code written for Python 3.* is guaranteed to work in all future versions.
- Both Python Version 2.* and 3.* are used currently.
- This course covers Python 3.*, but it isn't hard to change from one version to another.
- Python has several different implementations … we will use the Intel Python 3

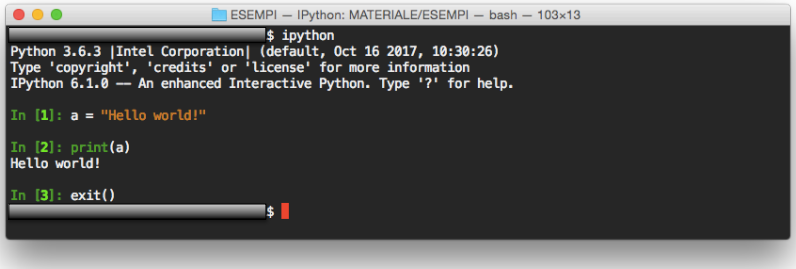


Running Python

- Interactive mode in a Python shell:

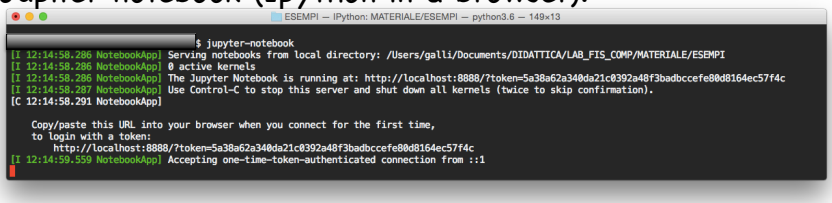


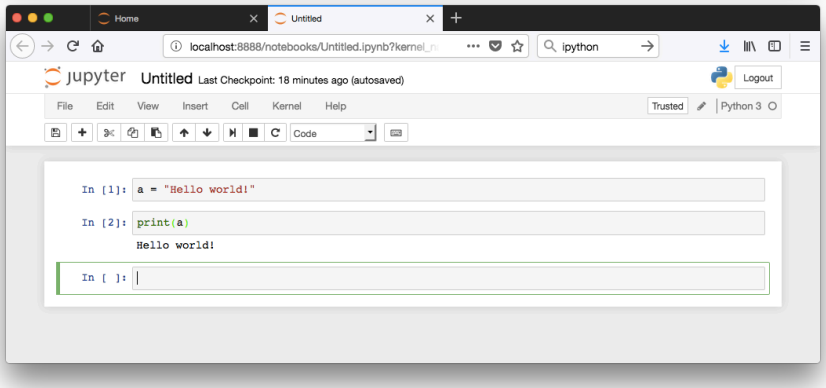
- Interactive mode with IPython:



Running Python

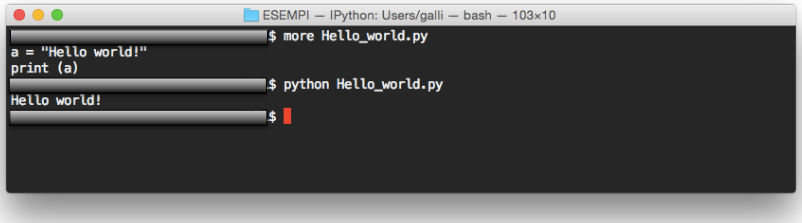
- Jupiter notebook (Ipython in a browser):



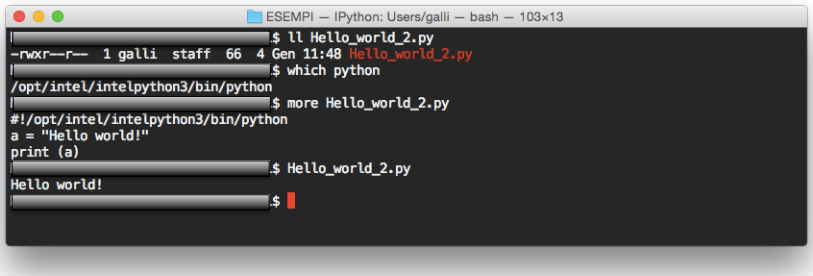


Running Python

- Use of a script:



- Turn your python script into a unix script:



Your First Program

- Let's start off by creating a short program that displays "Hello world!".
- In Python, we use the **print** statement to output text:

```
>>> print('Hello world!')
Hello world!
```

- Congratulations! You have written your first program ...
- Note the >>> in the code above. They are the prompt symbol of the Python console/shell. Python is an interpreted language, which means that each line is executed as it is entered.
- To exit from the Python console type:

```
>>> exit()
```

Simple operations

- Python has the capability of carrying out calculations.
- Enter a calculation directly into the Python console, and it will output the answer.

```
>>> 5 + 4 - 3
6
```

- Python also carries out multiplication and division, using an asterisk to indicate multiplication and a forward slash to indicate division.
- Use parentheses to determine which operations are performed first.

```
>>> 2 * (3 + 4)
14
>>> 10 / 2
5.0
```

Simple operations

- The minus sign indicates a negative number.

```
>>> (-7 + 2) * (-4)
20
```

- Dividing by zero in Python produces an error, as no answer can be calculated.

```
>>> 11/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- In Python, the last line of an error message indicates the error's type.
- Read error messages carefully, as they often tell you how to fix a program/script!

Floats

- Floats are used in Python to represent numbers that aren't integers. Some examples of numbers that are represented as floats are 0.5 and -7.8237591.
- They can be created directly by entering a number with a decimal point, or by using operations such as division on integers. Extra zeros at the number's end are ignored.

```
>>> 3/4
0.75
>>> 9.8765000
9.8765
```

- Computers can't store floats perfectly accurately, in the same way that we can't write down the complete decimal expansion of 1/3 (0.3333333333333333...). Keep this in mind, because it often leads to infuriating bugs!

Floats

- A float is also produced by running an operation on two floats, or on a float and an integer.

```
>>> 6 * 7.0
42.0
>>> 4 + 1.65
5.65
```

- A float can be added to an integer, because Python silently converts the integer to a float.
- However, this implicit conversion is the exception rather the rule in Python - usually you have to convert values manually if you want to operate on them.

Exponentiation

- Besides addition, subtraction, multiplication, and division, Python also supports exponentiation, which is the raising of one number to the power of another. This operation is performed using two asterisks.

```
>>> 2**5
32
>>> 9 ** (1/2)
3.0
```

Quotient & Remainder

- To determine the quotient and remainder of a division, use the floor division and modulo operators, respectively
- Floor division is done using two forward slashes
- The modulo operator is carried out with a percent symbol (%)

```
>>> 20 // 6
3
>>> 1.25 % 0.5
0.25
```

- These operators can be used with both floats and integers

Strings

- If you want to use text in Python, you have to use a string. A string is created by entering text between two single or double quotation marks.
- When the Python console displays a string, it generally uses single quotes. The delimiter used for a string doesn't affect how it behaves in any way

```
>>> "Python is fun!"
'Python is fun!'
>>> 'Always look on the bright side of life'
'Always look on the bright side of life'
```

- Some characters can't be directly included in a string. For instance, double quotes can't be directly included in a double quote string; this would cause it to end prematurely.

Strings

- Characters like these must be escaped by placing a backslash before them.
- Other common characters that must be escaped are newlines and backslashes.
- Double quotes only need to be escaped in double quote strings, and the same is true for single quote strings.

```
>>> 'Brian\'s mother: He\'s a naughty boy!'
'Brian's mother: He's a naughty boy!'
```

- Backslashes can also be used to escape tabs (**\t**), arbitrary Unicode characters, and various other things that can't be reliably printed. These characters are known as escape characters.

Strings

- **\n** represents a new line
- Python provides an easy way to avoid manually writing **\n** to escape newlines in a string. Create a string with three sets of quotes, and newlines that are created by pressing Enter are automatically escaped for you.

```
>>> """Customer: Good morning.
Owner: Good morning, Sir. Welcome to the
National Cheese Emporium."""
'Customer: Good morning.\nOwner: Good morning,
Sir. Welcome to the National Cheese Emporium.'
```

- As you can see, the **\n** was automatically put in the output, where we pressed Enter.

Output

- Usually, programs take input and process it to produce output.
- In Python, you can use the **print** function to produce output. This displays a textual representation of something to the screen.

```
>>> print(1 + 1)
2
>>> print("Hello\nWorld!")
Hello
World!
```

- When a string is printed, the quotes around it are not displayed.

Input

- To get input from the user in Python, you can use the intuitively named **input** function.
- The function prompts the user for input, and returns what they enter as a **string** (with the contents automatically escaped).

```
>>> input("Enter something please: ")
Enter something please: This is what\nthe user enters!
'This is what\\nthe user enters!'
```

- The print and input functions aren't very useful at the Python console, which automatically does input and output. However, they are very useful in actual programs and scripts.

Concatenation

- As with integers and floats, strings in Python can be added, using a process called concatenation, which can be done on any two strings.
- When concatenating strings, it doesn't matter whether they've been created with single or double quotes.

```
>>> "Spam" + 'eggs'
'Spameggs'

>>> print("First string" + ", " + "second string")
First string, second string
```

Concatenation

- Even if your strings contain numbers, they are still added as strings rather than integers. Adding a string to a number produces an error, as even though they might look similar, they are two different entities.

```
>>> "2" + "2"
'22'
>>> 1 + '2' + 3 + '4'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- In future slides, only the final line of error messages will be displayed, as it is the only one that gives details about the type of error that has occurred.

String Operations

- Strings can also be multiplied by integers. This produces a repeated version of the original string. The order of the string and the integer doesn't matter, but the string usually comes first.

```
>>> print("spam" * 3)
spamspamspam
>>> 4 * '2'
'2222'
```

- Strings can't be multiplied by other strings. Strings also can't be multiplied by floats, even if the floats are whole numbers.

```
>>> '17' * '87'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'pythonisfun' * 7.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Type Conversion

- In Python, it's impossible to complete certain operations due to the types involved. For instance, you can't add two strings containing the numbers 2 and 3 together to produce the integer 5, as the operation will be performed on strings, making the result '23'.
- The solution to this is type conversion.
- In the following example, you would use the **int** function.

```
>>> "2" + "3"
'23'
>>> int("2") + int("3")
5
```

- In Python, the types we have used so far have been integers, floats, and strings. The functions used to convert to these are **int**, **float** and **str**, respectively.

Type Conversion

- Another example of type conversion is turning user input (which is a string) to numbers (integers or floats), to allow for the performance of calculations.

```
>>> float(input("Enter a number: ")) +
      float(input("Enter another number: "))
Enter a number: 40
Enter another number: 2
42.0
```

- What is the output of this code?

```
>>> float("210" * int(input("Enter a number:" )))
Enter a number: 2
```

- Answer: 210210.0

Variables

- Variables play a very important role in most programming languages, and Python is no exception. A variable allows you to store a value by assigning it to a name, which can be used to refer to the value later in the program.
- To assign a variable, use one equals sign. Unlike most lines of code we've looked at so far, it doesn't produce any output at the Python console.

```
>>> x = 7
>>> print(x)
7
>>> print(x + 3)
10
```

- You can use variables to perform corresponding operations, just as you did with numbers and strings. As you can see, the variable stores its value throughout the program.

Variables

- Variables can be reassigned as many times as you want, in order to change their value.
- In Python, variables don't have specific types, so you can assign a string to a variable, and later assign an integer to the same variable.

```
>>> x = 123.456
>>> print(x)
123.456
>>> x = "This is a string"
>>> print(x + "!")
This is a string!
```

Variable names

- Certain restrictions apply in regard to the characters that may be used in Python variable names. The only characters that are allowed are letters, numbers, and underscores. Also, they can't start with numbers.
- Not following these rules results in errors.

```
>>> this_is_a_normal_name = 7

>>> 123abc = 7
SyntaxError: invalid syntax

>>> spaces are not allowed
SyntaxError: invalid syntax
```

- Python is a case sensitive programming language. Thus, **`Lastname`** and **`lastname`** are two different variable names in Python.

Variables

- Trying to reference a variable you haven't assigned to causes an error.

```
>>> foo = "a string"
>>> foo
'a string'
>>> bar
NameError: name 'bar' is not defined
```

- You can use the **del** statement to remove a variable, which means the reference from the name to the value is deleted, and trying to use the variable causes an error. Deleted variables can be reassigned to later as normal.

```
>>> del foo
>>> foo
NameError: name 'foo' is not defined
```

Variables

- You can also take the value of the variable from the user input.

```
>>> foo = input("Enter a number: ")
Enter a number: 7
>>> print(foo)
7
```

- … but, of course, no check is done on the input type:

```
>>> foo = input("Enter a number: ")
Enter a number: dog
>>> print(foo)
dog
```

In-place operators

- In-place operators allow you to write code like `x = x + 3` more concisely, as `x += 3`
- The same thing is possible with other operators such as -, *, / and % as well.

```
>>> x = 2
>>> x += 3
>>> print(x)
5
```

These operators can be used on types other than numbers, as well, such as strings.

```
>>> x = "spam"
>>> x += "eggs"
>>> print(x)
spameggs
```

- Many other languages have special operators such as `++` as a shortcut for `x += 1` . Python does not have these.

Using an editor

- So far, we've only used Python with the console, entering and running one line of code at a time.
- Actual programs/scripts are created differently; many lines of code are written in a file, and then executed with the Python interpreter.
- Python source files have an extension of **`.py`**
- As seen in the first examples, Python programs/scripts can be executed at the shell prompt by entering:

```
python script.py
```

Booleans

- Another type in Python is the Boolean type. There are two Boolean values: **`True`** and **`False`**.
- They can be created by comparing values, for instance by using the equal operator `==`

```
>>> my_boolean = True
>>> my_boolean
True
>>> 2 == 3
False
>>> "hello" == "hello"
True
```

- Be careful not to confuse assignment (one equals sign) with comparison (two equals signs).

Comparison

- Another comparison operator, the not-equal operator `!=` , evaluates to **`True`** if the items being compared aren't equal, and **`False`** if they are

```
>>> 1 != 1
False
>>> "eleven" != "seven"
True
>>> 2 != 10
True
```

- Python also has operators that determine whether one number (float or integer) is greater-than or smaller-than another. These operators are `>` and `<` respectively.

```
>>> 7 > 5
True
>>> 10 < 10
False
```

Comparison

- The greater-than-or-equal-to, and smaller-than-or-equal-to operators are >= and <=
- They return **True** when comparing equal numbers.

```
>>> 7 <= 8
True
>>> 9 >= 9.0
True
```

- Greater-than and smaller-than operators can also be used to compare strings lexicographically (the alphabetical order of words is based on the alphabetical order of their component letters).

```
>>> 'a' < 'b'
True
>>> 'professor' > 'student'
False
```

if statements & indentation

- You can use **if** statements to run code if a certain condition holds.
- If an expression evaluates to **True**, some statements are carried out. Otherwise, they aren't carried out.
- An **if** statement looks like this:

```
if expression:
    statements
```

- Python uses indentation (white space at the beginning of a line) to delimit blocks of code.
- Other languages, such as C, use curly braces to accomplish this, but in Python indentation is mandatory; programs won't work without it. As you can see, the statements in the if should be indented

if statements

- Here is an example **if** statement:

```
if 10 > 5:
   print("10 greater than 5”)
print("Program ended")
```

The expression determines whether 10 is greater than five. Since it is, the indented statement runs, and "10 greater than 5" is output. Then, the unindented statement, which is not part of the **if** statement, is run, and "Program ended" is displayed. Result:

```
>>>
10 greater than 5
Program ended
>>>
```

Notice the colon at the end of the expression in the **if** statement.

if statements

- To perform more complex checks, **if** statements can be nested, one inside the other.
- This means that the inner **if** statement is the statement part of the outer one. This is one way to see whether multiple conditions are satisfied. For example:

```
num = 12
if num > 5:
   print("Bigger than 5")
   if num <= 47:
      print("Between 5 and 47”)
```

Result:

```
>>>
Bigger than 5
Between 5 and 47
>>>
```

else statements

- An **else** statement follows an **if** statement, and contains code that is called when the **if** statement evaluates to **False**.
- As with **if** statements, the code inside the block should be indented.

```
x = 4
if x == 5:
   print("Yes")
else:
   print("No")
```

Result:
```
>>>
No
>>>
```

else statements

- You can chain **if** and **else** statements to determine which option in a series of possibilities is true. For example:

```
num = 7
if num == 5:
  print("Number is 5")
else:
  if num == 11:
    print("Number is 11")
  else:
    if num == 7:
      print("Number is 7")
    else:
      print("Number isn't 5, 11 or 7")
```

Result:
```
>>>
Number is 7
>>>
```

elif statements

- The **elif** (short for else if) statement is a shortcut to use when chaining **if** and **else** statements. A series of **if elif** statements can have a final **else** block, which is called if none of the **if** or **elif** expressions is **True** .

```
num = 7
if num == 5:
   print("Number is 5")
elif num == 11:
   print("Number is 11")
elif num == 7:
   print("Number is 7")
else:
   print("Number isn't 5, 11 or 7")
```

Result:

```
>>>
Number is 7
>>>
```

Boolean logic

- Boolean logic is used to make more complicated conditions for **if** statements that rely on more than one condition.
- Python's Boolean operators are **and**, **or**, and **not**.
- The **and** operator takes two arguments, and evaluates as **True** if, and only if, both of its arguments are **True**. Otherwise, it evaluates to **False**.

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 and 2 == 3
False
>>> 1 != 1 and 2 == 2
False
>>> 2 < 1 and 3 >  6
False
```

- Python uses words for its Boolean operators, whereas most other languages use symbols such as &&, || and !

Operator Precedence

- Operator precedence is a very important concept in programming. It is an extension of the mathematical idea of order of operations (multiplication being performed before addition, etc.) to include other operators, such as those in Boolean logic.
- The below code shows that == has a higher precedence than or:

```
>>> False == False or True
True
>>> False == (False or True)
False
>>> (False == False) or True
True
```

- Python's order of operations is the same as that of normal mathematics: parentheses first, then exponentiation, then multiplication/division, and then addition/subtraction.

- The following table lists all of Python's operators, from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

while loops

- The statements inside a **while** statement are repeatedly executed, as long as the condition holds. Once it evaluates to False, the next section of code is executed. The code in the body of a while loop is executed repeatedly. This is called iteration
- Below is a while loop containing a variable that counts up from 1 to 5, at which point the loop terminates.

```
i = 1
while i <=5:
    print(i)
    i = i + 1

print("Finished!")
```

Result:

```
>>>
1
2
3
4
5
Finished!
>>>
```

while loops

- The infinite loop is a special kind of **while** loop; it never stops running. Its condition always remains True.
- An example of an infinite loop:

```
while 1==1:
  print("In the loop")
```

- This program would indefinitely print "In the loop"
- You can stop the program's execution by using the **Ctrl-C** shortcut or by closing the program.

break

- To end a while loop prematurely, the **break** statement can be used.
- When encountered inside a loop, the break statement causes the loop to finish immediately.

```
i = 0
while 1==1:
  print(i)
  i = i + 1
  if i >= 5:
    print("Breaking")
    break

print("Finished")
```

Result:

```
>>>
0
1
2
3
4
Breaking
Finished
>>>
```

- Using the break statement outside of a loop causes an error.

continue

- Another statement that can be used within loops is **continue**. Unlike break, **continue** jumps back to the top of the loop, rather than stopping it
- Basically, the **continue** statement stops the current iteration and continues with the next one.

```
i = 0
while True:
   i = i +1
   if i == 2:
      print("Skipping 2")
      continue
   if i == 5:
      print("Breaking")
      break
   print(i)
print("Finished")
```

Result:

```
>>>
1
Skipping 2
3
4
Breaking
Finished
>>>
```

- Using the continue statement outside of a loop causes an error.

Suggested books & material

Hans Petter Langtangen:
- *A Primer on Scientific Programming with Python*
- *Python Scripting for Computational Science*

51