1 How to use python : A first example

• Interactive mode programming - Open a shell and type:

```python
python
```
This opens the interpreter. Type then your first instruction (or statement) after the prompt (>>>):

```python
print "Hello World"
```
Here is another way:

```python
message = "Hello world"
p
```n
• Script mode programming - In your folder /course/learnpython, create a text file named "test.py". Write these two last lines again in this file. SAVE ! Create a new shell, go to the same folder and type python test.py.

2 Variables

• A variable is a way to store some data that can vary. Each variable has a name denoting the store location and a value which is the data stored.

• Declaration: to declare a variable is to announce to the computer the name and the existence of a new variable (ie creating the blue box). With python, a variable is declared automatically the first time you use it. For other programming languages it is sometimes more complicated.

• Assignment: to assign a value to a variable is to set or re-set the exact data that you want to store (ie storing the data in the box).
Let’s go back to the first example:
In this single line: `message = "Hello world"`, we have:
- declared a new variable named `message`,
- assigned `message` the value "Hello world",
- In the next line we’ve asked the computer to print the value of `message`.

Now let’s change the value of `message` (reassignment) and print the new value:
```
message = "Bonjour"
print message
```
Note that the first value "Hello world" is now lost!

3 Data type

**Example**: What happens with this?
```
a = 2
b = "b"
print a
print b
print a+a
print b+b
print a+b
```

- Every variable can be seen as an object with a *type* defining the nature of its value: a number, a string of characters, a list of objects ...
- Mixing operators between numbers and strings (ie print a+b) is not supported.

**Primitive data types**:
- **Integer (int)**: `a = 3`
- **Floating-point number (float)**: `b = 3.0` or `b = float(3)`
  - Python can apply basic arithmetic operators on numbers:
    ```
    (a+b)/2
    float(a)/2
    ```
- **String (str)**: `print "a : " + str(a)`
  - We can access part of a string using the indexing operator (note that the first index is 0!):
    ```
    s = "stringExample"
    print s[0]
    print s[3], s[-1]
    ```
  - "\n" stands for a new line and "\t" for a tab.
  - Try to type `print "Hello\world"` and print "Hello\nworld".
- **Boolean**: logical values `True` and `False`

Now try to create a variable and print it to read:
```
This
is
OK!
```
Complex data types:
Apart from these primitive data types, all other data types are more complex and can store structured set of values. In the next section we introduce two examples: tuples and lists. We will see how copying variables is different depending on the data type.

4 Tuples and lists

- Python tuples are ordered sequences of any Python objects, including other tuples.
- Creating a tuple is as simple as putting different comma-separated values, optionally between parentheses:
  ```python
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5);
tup3 = "a","b","c","d";
```
- Tuples are immutable, thus it not possible to change the value of an element of a tuple, or to delete or add an element to a tuple.
- To access values in tuple, use the square brackets with the indices (remember that indices start at 0):
  ```python
  print tup1[1]  # prints out "chemistry"
  ```
- Python lists are, like tuples, ordered sequences of any Python objects, including other lists, or tuples. Lists are created with brackets and elements are accessed the same way than for tuples:
  ```python
  mylist = ["a","b","c"]
  print mylist[1:3]  # prints out ['b', 'c']
  ```
- The big difference with tuples is that lists are mutables. You can insert, delete and replace elements in a list.

```
Example:
1 = []
1.append(1)
1.append("text")
print 1
print1[0])  # prints 1
print1[1]  # prints text
1.remove(1)
len(1)  # prints 2
```

Comments: all characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them.

- Note that a string is very similar to a list of characters:

```
>>> seq = 'agcgcccttgatt'
>>> print seq[:3], seq[3:6], seq[-3:]
agc gcc att
```
However, while lists are *mutables* strings are not, which means that it is not possible to add, change or remove one character of a string without redefining the whole string.

- As for other mutables data types, and contrary to immutables data types like primitive types or tuples, copying a list in a second variable will only copy the *reference* to this list, which means that the list is still unique:

  ```
  >>> l1 = [1, 2, 3]
  >>> l2 = l1  # copying the reference to the list only !
  >>> l1[0] = 4  # the list is modified through l1
  >>> l2  # this is still the same list
  [4, 2, 3]
  ```

- It is possible to nest lists (create lists containing other lists), for example:

  ```
  >>> q = [2, 3]
  >>> p = [1, q, 4]
  >>> len(p)
  3
  >>> p[1]
  [2, 3]
  >>> p[1][0]
  2
  >>> p[1].append('xtra')
  >>> p
  [1, [2, 3, 'xtra'], 4]
  >>> q
  [2, 3, 'xtra']
  ```

  Note that here `p[1]` and `q` really refer to the same object.

### Exercise: what is the result of this following code? Why?

```python
l1=[1,2,3]
l2=[l1[0],l1[1],l1[2]]
l1[0]=4
l2
```

### 5 Conditions

- Python uses boolean variables to evaluate conditions.
- The boolean values True and False are returned when an expression is compared or evaluated.
- The standard comparison operators are: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to)

  ```
  >>> x=2  # assign x to 2
  >>> print x==2  # ask if the value of x is 2, prints as True
  >>> print x==3  # prints out False
  ```
• **if** and **else** statements: the **if** statement defines a block of code that is executed if the condition is true. The optional **else** statement contains the code that executes if the condition is false.

```python
>>> if x==2:
...    print "x=2"
>>> else:
...    print "x:"+x
```

Note that the block of code contained in the **if** or **else** statement is indented: *indentation* is Python's way of grouping statements.

• The **elif** statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.

```python
>>> if x==2:
...    print "x equals 2"
>>> elif x>2:
...    print "x greater than 2"
>>> else:
...    print "x smaller than 2"
```

• **Boolean operators:** the **and** and **or** boolean operators allow building complex boolean expressions, for example:

```python
>>> seq1 = 'acgccttgaa'
>>> if seq1[0] == 'a' and seq1[1] == 'c':
...    print "The sequence starts with 'ac'."
```

• The **in** operator can be used to check if a specified object exists within an iterable object container, such as a list:

```python
>>> if 'a' in seq1:
...    print "The sequence contains an adenine base."
```

• Using **not** before a boolean expression inverts it.

### 6 Loops

A loop statement allows us to execute a statement or group of statements multiple times.

• Like for conditions, the body of the loop is indented.

• **While loop:** repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```python
>>> a, b = 0, 1  # Multiple assignment
>>> while b < 10:
...    print b
...    a, b = b, a+b
...    print b
1
```
• For loop: executes a sequence of statements multiple times.

– First example:

```python
>>> for letter in 'Python':
...    print 'Current Letter :', letter
```

– Second example: iterating by sequence index

```python
>>> bases = ['A', 'C', 'G', 'T']
>>> for index in range(len(bases)):
...    print 'Current base :', bases[index]
```

Example of nested loops:

```python
for num in range(10,20):  # to iterate between 10 to 20
    for i in range(2,num): # to iterate on the factors of the number
        if num%i == 0:   # to determine the first factor
            j=num/i      # to calculate the second factor
            print '%d equals %d * %d' % (num,i,j)
        break       # to move to the next number, the first FOR
    else:            # else part of the loop
        print num, 'is a prime number'
```

This code is written in ”/course/docspython/loop.py”, try to execute it !

Exercise on loops

Solve each exercise by writing the solution code in a text file (e.g. loops_ex1.py) in your folder /course/learnpython, and running the script to check.

1 - Write the sequence 'atgcggctgtgatgtagtcctttc' into a string variable (you can also copy it from /course/docspython/variables.py”).

Write a loop going through the sequence and counting the numbers of 'g' and 'c'.

Finally compute the GC% of the sequence.

Run your script.

2 - Define a list of 10 integers of your choice so that the order of the elements appears random. Use a loop to store into two variables the minimum and the maximum values of this list.

3 - Find the Greatest Common Divisor of two integers p and q of your choice (for example 195 and 117), following this procedure:

   • Loop until p = q :

     – If p < q then swap p and q
– If \( p > q \) then substract \( q \) from \( p \)

• Print \( p \)

7 Functions

• A function is a block of code that is named so it can be reusable. Python gives you many built-in functions like `print()`, `len()` etc.

• A function can take input parameters called arguments that have to be of adequate type (e.g. in `print("Hello")`). It can also return an object (simple or complex data type).

• You can create your own functions. These functions are called user-defined functions.
  – Function blocks begin with the keyword `def` followed by the function name and parentheses. Any input parameters or arguments should be placed within these parentheses.
  – The code block within every function starts with a colon (:) and is indented.
  – The statement `return [expression]` exits a function, optionally passing back an expression to the caller.
  – Parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back outside the function. For example:

```python
# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist.append([1,2,3,4]);
    return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "List: ", mylist
```

However, this is not true for parameters whose type is immutable (numbers, strings, tuples). The changes made on the parameters inside the function won’t reflect outside the function.

• Global vs. Local variables: Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared whereas global variables can be accessed throughout the program body by all functions.

What happens in this example?

```python
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2; # Here total is local variable.
```
print "Inside the function local total : ", total
return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total

This produces the following result :

Inside the function local total : 30
Outside the function global total : 0

**Question: how you do change the value of total outside the function?**

→ One way is to add the statement `global total` at the beginning of the function `sum`. This way the variable `total` used in the function is not a local variable.

→ What other way can you find?

**Methods:** methods are functions associated to a particular object (list, string, etc), and have to be called with reference to one of this object. For example `append()` and `remove()` are methods for lists:

```python
>>> seqlist = ["actg", "gaattc", "attagc"] # seqlist is a variable of type list
>>> seqlist.append("tgcca") # append is called for seqlist
```

**Exercises on functions**

1 - Define a function `reverse()` that computes the reversal of a string. For example, `reverse("I am testing")` should return the string "gnitset ma I".

2 - Define a function `fact()` taking in argument an integer and returning its factorial.

3 - Define a function `is_palindrome()` that recognizes palindromes (i.e. words that look the same written backwards). For example, `is_palindrome("radar")` should return `True`.

8 **Dictionaries**

- A dictionary is mutable and is another container type that can store any number of Python objects, including other container types. Dictionaries consist of pairs (called items) of keys and their corresponding values.

- Python dictionaries are also known as associative arrays or hash tables.

- Unlike lists, dictionaries are unsorted collections.

- In a dictionary each key is unique and can be a string, a number or a tuple. The value can be any object.

- To access dictionary elements, you can use the square brackets along with the key to obtain its value.

```python
>>> mydict = {'Gene': 'Bmal1', 'Protein': 'BMAL1', 'Expr': 1.2};
>>> print "mydict['Gene']": ", mydict['Gene'];
```
• You can update a dictionary by adding a new entry or item (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry as shown below in the simple example:

```python
>>> mydict = {'Gene': 'Bmal1', 'Protein': 'BMAL1', 'Expr': 1.2}
>>> mydict['Expr'] = 2  # update existing entry
>>> mydict['Chromosome'] = 11  # Add new entry
>>> del mydict['Protein']  # remove entry with key 'Protein'
>>> mydict.clear()  # remove all entries in mydict
>>> del mydict  # delete entire dictionary
```

• Some useful dictionary functions and methods:
  
  - `cmp(dict1, dict2)`: Compares elements of both dictionaries.
  - `len(dict)`: Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
  - `str(dict)`: Produces a printable string representation of a dictionary.
  - `type(variable)`: Returns the type of the passed variable.
  - `dict.copy()`: Returns a shallow copy of dictionary `dict`.
  - `dict.fromkeys(seq,value)`: Creates a new dictionary with keys from `seq` and values set to `value`.
  - `dict.get(key, default=None)`: For key `key`, returns value or default if `key` not in dictionary.
  - `dict.has_key(key)`: Returns `True` if `key` in dictionary `dict`, `False` otherwise.
  - `dict.items()`: Returns a list of `dict`'s (key, value) tuple pairs.
  - `dict.keys()`: Returns a list of dictionary `dict`'s keys.
  - `dict.setdefault(key, default=None)`: Similar to `get()`, but will set `dict[key]=default` if `key` is not already in `dict`.
  - `dict.update(dict2)`: Adds dictionary `dict2`'s key-values pairs to `dict`.
  - `dict.values()`: Returns a list of dictionary `dict`'s values.

### Exercise on dictionaries

1. Write a function computing the codon usage of a sequence, that is the frequency of each codon in this sequence. You will use for this a dictionary with codons as the keys and the number of usage as values. Here is the algorithm:
   
   For each codon in the coding sequence:
   - if the codon is already in the dictionary of the usage, then add 1 to the count
   - otherwise, put the codon in the dictionary with `count = 1`

2. Let `code` be this dictionary:

   ```python
code = {
   'ttt': 'F', 'tct': 'S', 'tat': 'Y', 'tgt': 'C',
   'ttc': 'F', 'tcc': 'S', 'tac': 'Y', 'tgc': 'C',
   'ttg': 'L', 'tcg': 'S', 'tag': '*', 'tgg': 'W',
   'ctt': 'L', 'cct': 'P', 'cat': 'H', 'cgt': 'R',
   'ctc': 'L', 'ccc': 'P', 'cac': 'H', 'cgc': 'R',
   'cta': 'L', 'cca': 'P', 'caa': 'Q', 'cga': 'R',
   'ctg': 'L', 'ccg': 'P', 'cag': 'Q', 'cgg': 'R',
   'act': 'T', 'acc': 'T', 'aac': 'N', 'agc': 'S',
   'agt': 'T', 'agg': 'R', 'aag': '*', 'agc': 'S',
   'agt': 'T', 'agg': 'R', 'aag': '*', 'agc': 'S'}
```
Using this dictionary (you can copy it from /course/docspython/variables.py), define a function which translates a string representing a coding DNA sequence into the corresponding protein.

9 Modules and Packages

A module is a component providing Python definitions of functions, variables or classes, all corresponding to a given specific theme. All these definitions are contained in a single Python file. Thanks to modules, you can reuse ready-to-use definitions in your own programs.

There are two ways to use a module with the import statement. For example, let’s say we want to use the built-in function getcwd() from the module os, that returns the current working directory. We can use it like this:

- >>> import os
  >>> os.getcwd()
  import os allows you to use any function of os with the prefix os.

- >>> from os import getcwd
  >>> getcwd()
  This allows you to use the function getcwd() only, without prefix.

- >>> from os import *
  >>> getcwd()
  This allows you to use any function of os without prefix. This should be avoided as it can lead to conflicts: if you import two modules sharing some function names, or if one of your variable or function is overwritten by the module.

Some useful modules:

- **math**: provides access to mathematical functions like math.exp(x), math.pow(x,y), math.sqrt(x), math.floor(x)...

- **numpy**: package for scientific computing with Python. It contains among other things tools to manipulate large multi-dimensional arrays, useful linear algebra, Fourier transform, and random number capabilities.

- **os**: provides a way of using operating system dependent functionality. For example os.listdir(path) and os.mkdir(path) are the equivalents to ls and mkdir in shell, while os.system(command) executes directly a shell command.
• **RPy**: Python interface to the R Programming Language.

• **Biopython**: a set of freely available Python tools for bioinformatics and molecular biology. Its features include:
  – Parsing bioinformatics files into python structures
  – A sequence class to store sequences, ids and features
  – Interface to popular bioinformatics programs (clustalw, blast, primer3 and more)
  – Tools for performing common operations on DNA/protein sequence (translation, transcription, Tm, weight)
  – Code to deal with alignments

**Example of Biopython’ sequence class use:**

```python
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import DNAAlphabet
>>> seq_1 = Seq('GATCGATGGCCTATATAGGA', DNAAlphabet())
>>> rna_1 = seq_1.transcribe()
>>> str(rna_1)
'GAUCGAUGGGCCUAUAUAGGA'
>>> rna_1.translate()
Seq('DRWAYIG', ExtendedIUAPCProtein())
```

**Exercise on modules**

Here you will use:

– the function `randn(n)` from the module `numpy.random`, that generates n gaussian random numbers

– the function `arange(a,b,c)` from the module `numpy`. This function create a sequence a sequence of numbers from `a` to `b` with a step `c`, as an array.

– From the module `math`, the variable `pi` and the function `sin(x)`.

– the functions `hist(x)` and `plot(x,y)` from the module `matplotlib.pyplot`. These function plot respectively the histogram of the vector `x`, and `y` vs `x`.

Generate a vector of 10000 gaussian random numbers and plot it as a histogram.
Then generate a vector named containing 200 points between 0 and 2. Using this vector, plot the values of sinus between 0 and 4Pi.
Import the function `savefig` from the module `matplotlib.pyplot` and save the plot with `savefig("sin.png")`. Check that the image is correctly created in your working directory.

**10 Files I/O**

Here we will see what are the basic input/output functions available in Python.

• **Standard output**: you already know the simplest way to produce output : with the `print` statement, where you can pass zero or more expressions, separated by commas.

• **Standard input**: the `raw_input("prompt")` function reads one line from standard input (keyboard) and returns it as a string:
str = raw_input("Enter your input: ");
print "Received input is : ", str

Exercise on standard input/output

Write a program able to play the "Guess the number"-game, where the number to be guessed is randomly chosen between 1 and 20. This is how it should work when run in a terminal:

```python
>>> import guess_number
Hello! What is your name?
Gandalf
Well, Gandalf, I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
Take a guess.
18
Good job, Gandalf! You guessed my number in 3 guesses!
```

Help: you can use the function `randint(n,m)` from the module `random` which returns a random integer between `n` and `m`.

- Opening and closing files: Before you can read or write a file, you have to open it using Python's built-in `open()` function. This function creates a file object which would be utilized to call other support methods or attributes associated with it.

```python
file myfile = open("filename.txt","w")       # Opens filename.txt or creates it
print "Name of the file: ", myfile.name
print "Closed or not : ", myfile.closed
print "Opening mode : ", myfile.mode
myfile.close()
```

Notice that in addition to the name of the file, you have to provide the mode in which the file has to be opened i.e. read (r), write (w), append (a) etc. The `close()` method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

- Reading files:
  - The `read()` method reads any string from an open file, for a specified number of bytes or until the end of the file.

```python
fo = open("/tmp/foo.txt", "r")  # Open a file with r+ mode (read+write)
str = fo.read(10)              # Read 10 bytes
print "Read String is : ", str
str2 = fo.read()               # Read the rest of the file
fo.close()                     # Close opend file
```

- `readline()` reads one line at a time.
• **Writing files**: Two modes are possible for writing: "w" (write) overwrites an existing file, while "a" (append) will write at the end of the file. Then the `write()` method writes any string to an open file.

• **File positions**:
  - The `tell()` method tells you the current position within the file. In other words, the next read or write will occur at that many bytes from the beginning of the file.
  - The `seek(offset[, from])` method changes the current file position. The `offset` argument indicates the number of bytes to be moved. The optional `from` argument specifies the reference position from where the bytes are to be moved (0: the beginning of the file (default), 1: the current position, 2: the end of the file).

• **Managing files and directories with the `os` module**:

```python
import os
os.rename("current_file_name", "new_file_name") # rename a file
os.remove("file_name") # remove a file
os.getcwdu() # get the current directory
os.listdir('dirname') # list the directory content
os.mkdir("newdir") # create a directory
os.chdir("newdir") # change the current directory
os.rmdir('dirname') # delete an empty directory
```

### Exercises on handling files

1. Write a version of a palindrome recogniser that accepts a file name from the user, reads each line, and prints the line to the screen if it is a palindrome.

2. Write a program that given a text file will create a new text file in which all the lines from the original file are numbered from 1 to n (where n is the number of lines in the file).

---

11 **Regular expressions**

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

In Python, regular expressions are used with the `re` module.

**Example**

`'[AP]{1,2}D'` is a regular expression that matches: AD, PD, AAD, APD, PAD and PPD.
### Regular expression patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches beginning of line</td>
<td><code>^Python</code> matches &quot;Python&quot; at the start of a string or internal line</td>
</tr>
<tr>
<td>$</td>
<td>Matches end of line</td>
<td>a.c matches &quot;abc&quot;, &quot;adc&quot; etc</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character except newline</td>
<td><code>[...]</code> matches anything in brackets</td>
</tr>
<tr>
<td>[^...]</td>
<td>Matches any single character not in brackets</td>
<td><code>[Pp]ython</code> matches &quot;Python&quot; or &quot;python&quot;</td>
</tr>
<tr>
<td>\w</td>
<td>Matches word characters. Equivalent to [a-zA-Z0-9_]</td>
<td><code>ruby*</code> matches &quot;rub&quot; plus 0 or more ys</td>
</tr>
<tr>
<td>\d</td>
<td>Matches digits. Equivalent to [0-9]</td>
<td><code>ruby+</code> matches &quot;rub&quot; plus 1 or more ys</td>
</tr>
<tr>
<td>r*</td>
<td>Matches 0 or more occurrences of r</td>
<td><code>ruby?</code> matches &quot;rub&quot; or &quot;ruby&quot;</td>
</tr>
<tr>
<td>r+</td>
<td>Matches 1 or more occurrence of r</td>
<td><code>\d{3}</code> matches exactly 3 digits</td>
</tr>
<tr>
<td>r?</td>
<td>Matches 0 or 1 occurrence of r</td>
<td><code>\d{3,5}</code> matches 3, 4 or 5 digits</td>
</tr>
<tr>
<td>r{n}</td>
<td>Matches exactly n number of occurrences of r</td>
<td><code>\[Pp\]ython\&amp;\1erl</code> matches &quot;python&amp;perl&quot; or &quot;Python&amp;Perl&quot;</td>
</tr>
<tr>
<td>r{n,m}</td>
<td>Matches n or more occurrences of r</td>
<td>a(bc)* matches &quot;abcbe&quot; and not &quot;abce&quot;</td>
</tr>
<tr>
<td>r{n,m}</td>
<td>Matches at least n and at most m occurrences of r</td>
<td><code>([Pp]ython\&amp;\1erl</code> matches &quot;python&amp;perl&quot; or &quot;Python&amp;Perl&quot;</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>Matches either a or b</td>
</tr>
<tr>
<td>(re)</td>
<td>Groups regular expressions and remembers matched text</td>
<td></td>
</tr>
<tr>
<td>\1...\9</td>
<td>Matches nth grouped subexpression</td>
<td></td>
</tr>
</tbody>
</table>

- Regular expressions use the backslash character (`\`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals.
- The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'`. So `r"\n"` is a two-character string containing `\` and `\n`, while "\n" is a one-character string containing a newline.

### Some useful functions

- `re.search(pattern, string)` attempts to match RE `pattern` to `string`. The `re.search` function returns a `match` object on success, `None` on failure. We can use `group(num)` function of `search` object to get matched expression. If the pattern is matched several times in `string`, only the first match is found.

Suppose you want to find the email address inside the string : ‘purple alice-b@google.com monkey dishwasher’

```python
>>> str = 'purple alice-b@google.com monkey dishwasher'
>>> match = re.search(r'\w+@\w+', str)
>>> match.group()
'b@google'
```

The search does not get the whole email address in this case because the `\w` does not match the `-` or `.` in the address. We’ll fix this using the regular expression features below.

```python
>>> match = re.search(r'([\w-.]+)@([\w-.]+)', str)
>>> match.group()
alice-b@google.com
```

```python
>>> match.group(0)
alice-b@google.com
>>> match.group(1)
```
alice-b

```python
>>> match.group(2)
google.com
```

- `re.findall(pattern, string)` returns all non-overlapping matches of `pattern` in `string`, as a list of strings (or a list of tuples of strings if the pattern holds subgroups).

```python
>>> re.findall(r"([AB]C)-([DE][FG])","AC-DG-BC-EG")  # Pattern with two subgroups
[('AC', 'DG'), ('BC', 'EG')]
>>> re.findall(r"[AB]C-[DE][FG]","AC-DG-BC-EG")  # Pattern without subgroups
['AC-DG', 'BC-EG']
```

```python
>>> str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
>>> emails = re.findall(r'\[\w\.-\]+@[\w\.-\]+', str)
>>> emails
['alice@google.com', 'bob@abc.com']
```

- `re.sub(pattern, repl, string)` returns the string obtained by replacing all occurrences of `pattern` in `string` by the replacement `repl`.

### Exercises on regular expressions

1. **Some regular expressions** : what do they match ?
   - `r"^[a2-9tjqk]{5}$"`
   - `r".*(.).*\1"`
   - `r"(CG\w)|(AG(A|G)"`

2. **Write** a script which takes the file `mots.txt` as input and returns 4 lists with:
   - words beginning with 'acc'
   - words ending with 'er'
   - words with 'ss'
   - words with 'ss' and ending with 'er'

3. **Go on regexcrossword.com** and solve crosswords 3 and 4 for beginners.

### 12 More exercises

#### Counting words

- **Write a function** `count_words` taking as argument a file name, and returning a dictionary containing all the different words contained in the text with the number of times they are used. You could define an intermediate function to extract all the words of a line.

- **Write a function** `main` taking as arguments an input file name and an output file name. The function should use `count_words` to count the words in the input file, and write the result as a table in the output file. Test it on the file `alice_in_wonderland.txt` in the folder `docspython`.
Some functions: `string.lower`, `string.split`, `string.replace`

**Regex**

In the folder `docspython` are several html pages showing the most popular names given to babies in different years, ordered in tables.

- Write a function `ParseOneFile` taking as argument a page’s path and returning the information in the table as a list of tuples, each tuple containing the number of the row and the two names in the row. You should look at the source code of the html file to decide how to capture the information with a regex.

- Write a function `collect_data` applying `parseOneFile` to all the pages and storing the data in a dictionary, with the years as keys.

Some functions: `os.listdir`, `re.search`, `re.match`, `file.read`

**Euler method**

- Write a function taking as arguments the derivative of a mathematic function, the boundaries of an interval and a step, and an initial value, and implementing the Euler method to compute the function’s value on the interval.
  Define three mathematic derivative functions to test it.
  Create a figure where you will plot the result for the three functions on the same interval.

- Write a new version of this Euler method for a two-variables function, taking two derivatives in argument. Test it again with a plot, with each variable as an axis.

Some functions: `numpy.zeros`, `numpy.arange`, `matplotlib.pyplot.plot`, `matplotlib.pyplot.savefig`, `matplotlib.pyplot.figure`

**Dynamical system**

- Write a function `solveequation(a,b,c,d,x0,y0,T,dt,mini,maxi)` where the arguments `a,b,c,d` are the components of a matrix `A`, `x0` and `y0` are initial values, `T` is a simulation time, `dt` is a time step and `mini, maxi` are boundaries. Similarly to the previous exercise on Euler’s method, the function should simulate on a plot the trace of the two-variables vector `X` satisfying the differential system `X’=AX`. Test the function with:
  - `solveequation(1.,-2.,-1.,0.,-1,-1.4,100,0.1,-5,5)`
  - `solveequation(0.5,-1.,1.,-0.6,2.,1.7,100,0.1,-5,5)`
  - `solveequation(0.5,-1.,1.,-0.4,2.,1.7,100,0.1,-5,5)`

- Complete this function with a computing of the eigenvalues. Print the eigenvalues, and when they are positive plot the lines with the corresponding slopes on your figure. Plot also the isoclines corresponding to `x’=0` or `y’=0`.

16