Python Programming

Jean-Pierre Messager (jp@xiasma.fr)

27 of September, 2020 – version 1.0b

No commercial use without authorization

Download this course and exercises

This course is published on a git repository

- Sign in on https://framagit.org/
- Send me your id
- I may promote you as a *reporter* on this project
    - Read access to material and exercises files
    - Opening tickets
- There: https://framagit.org/jpython/python-course
- Others projects : https://framagit.org/jpython/meta
- You can access to updates and new versions
- For every major revision a new *tag* is created

Python Programming

Contents

- What is Python ?
- A first program
- Storing and processing data
- Collections
- Loops
- Functions
- Debugging and test
- Object Oriented Programming
- Modules
- A few words on Anaconda, NumPy, Pandas

Storing and processing data

- Names and references
- Data types
- Operators, functions and methods
- Booleans and conditions

Collections

- Data collections
- Lists
- Changing and extracting data
- Others type of collections

Loops

- Looping through lists
- Looping through other collections
- Comprehensions

Functions

- Structuring programs
- Passing arguments
- Returning values
- Function calls

Debugging and tests

- Interpreting error messages
- Debuggers
- Unit tests

Object Oriented Programming

- Defining new classes
- Constructors
- Normal and dunder methods
- Overloading operators
- Inheritance

Modules

- Importing custom modules
- Preventing code execution
- Special methods and attributes

History of Python

Python 1

- Created in 1989-1991 by the Dutch programmer Guido van Rossum
- Named as a tribute to the (mainly) British comedy troup *Monty Python*

Python 2

- Supported by the non-profit organization *Python Software Fundation*, created in 2008
- Is no more maintained since 1st of January 2020

Python 3

- Published in 2008: should be the version you use!
- Guido van Rossum stepped down as the PSF *Benevolent Dictator For Life* in 2018, but is still deeply involved in the language development

Programming languages paradigms

Several logical *paradigms*

- Imperative programming: bunch of expressions and instructions
  - C, Pascal, Fortran
- Functional programing: calling and returning *functions*
  - LISP, Scheme, CaML, Haskell
- Object Oriented programming: data and operations are encapsulated together
  - Java, C#, Smalltalk
- Some languages, like Python, allow various paradigms

Constraining data manipulation

- Déclarative: you have to specify all data types (integer, float, strings, arrays, . . .)
- Strongly typed: no (or almost no) implicit type conversion
- Python is strongly typed but not declarative

Programming languages

Low level languages

- Explicit memory allocation
- Compiled to machine code, then translated to binary
- Conceptually close to assembly language, but more convenient
- Very high performance
- Usually used to write operating systems, utilities, and system libraries
- High performance and real time computing, games
- To name a few: *C, C++, D, Fortran, Pascal, PL/1, Ada*

Other languages implementations (compilers, interpreters) are usually written in one of these languages.

Another way to consider programming languages

High–level mathematical languages

- Automatic memory allocation
- Usually interpreted (running on top of a *virtual machine*)
- Based on advanced abstract mathematical concepts
- Examples : *LISP, Scheme, CaML, Haskell*

High–level pragmatic languages

- Easier to deal with
- Could lead to bad practices and unmaintable code: *BASIC, PHP, JavaScript*
- Or not: *Ruby, Python*

This course is based on Python 3.

Python

Python's history

- Created in 1991 by Guido Van Rossum
- Focus on readibility, rigor, ease of use and expressiveness
- Fit for beginners *as well as* experienced programmers
- Python 2 gained popularity during the 200x
- Python 3 is about ten years old now
- Free Software *(Open Source)*

Features

- Object-oriented from the ground up
- Multiple paradigms : imperative, functional, object oriented
- Portable: UNIX, GNU/Linux, MS Windows, ...
- Very rich set of high level libraries (modules)

Instructions and expressions

Python makes a difference between *expressions* and *instructions*.

An expression compute something and *has a value*:

```
42 + 13
"Hello " + name + "!"
len(foodInfo)
```

Values: 55, 'Hello John!' (if *name* is 'John') then the length of data structure *foodInfo*.

An expression can have a *side effect*

i.e. "doing *something* in addition to have a value"

```
print('Hello!')
```

The value here is *None*, a special value in Python aimed to represent "nothingness".

Instructions and expressions

An instruction *does* something

An instruction only has a *side effect* it has *no* value!

```
res = res * i   # assignment
if price < 42: # this line and the next one are
    print("It's cheap!") # one instruction
```

An instruction can embed expressions

```
msg = 'Hello ' + name
if price > 12 and price < 56:
    print("Price ok.")
```

- `'Hello ' + name` is an expression
- `price > 12 and price < 56` too
- also `print(...)`

An expression *cannot* contain instructions.

Usual errors

Triggered by the compiler

If you write an instruction in a place where an expression is awaited, an *Exception* is *raised* at compilation time:

```
if i = 1:
...
SyntaxError: invalid syntax
```

Untriggered by the compiler (at best you'll get a *warning*)

Writing down an expression instead of an instruction : what you asked for is computed but has no practical effect : it is a BUG!

```
31 + 11
if price == 42:
    print('Good price!')
```

What was intended:

```
price = 31 + 11
```

Compiled languages

Compiler generate assembly code, then translate to binary

```
$ gcc -o factorial factorial.c
$ file factorial
factorielle: ELF 64-bit LSB pie executable,
x86_64, version 1 (SYSV), dynamically linked,
...
$ ./factorial
Number: 5
120
```

You can have a look to intermediate assembly code

```
$ gcc -S factorielle.c
$ less factorielle.s
```

Interpreted languages

Bytecode

- Most interpreted languages are compiled into a specific format: the *bytecode*
    - It's a binary format
    - Independant of any CPU architecture
    - Supposed to be executed by a specific software: a *virtual machine*
- Examples : Java, C#, Python, Perl, ...

Some virtual machines

- JVM : *Java Virtual Machine* (Java, Groovy, ...)
- .NET *Runtime* (C#, VisualBasic.NET, F#, ...)
- PVM : *Python Virtual Machine*
- Perl 6 : *Parrot*, *MoarVM*

Compiling and executing

Both steps can be separated:

```
$ javac hello.java
$ ls
hello.java hello.class
$ java hello
Hello World!
```

... or taking place in one go:

```
$ python3 hello.py
Hello World!
```

Bytecode

You can *disassemble* a program

In Python the *dis* module allows you to look at bytecode:

```
        ...
        >>    20 FOR_ITER                12 (to 34)
              22 STORE_FAST               2 (i)

  4           24 LOAD_FAST                1 (res)
              26 LOAD_FAST                2 (i)
              28 BINARY_MULTIPLY
              30 STORE_FAST               1 (res)
              32 JUMP_ABSOLUTE           20
        >>    34 POP_BLOCK
        ....
```

Libraries

Libraries extend the language

- Solve a specif problems or help you to solve it
- A library extends language features:
  - Mathematics
  - Text processing
  - Graphical User Interfaces, Web programming
  - Databases access
  - etc.
- Operating systems provides several binary libraries (`.dll` for MS Windows, `.so` for UNIX and GNU/Linux)
- You can then install more of them

Python libraries

Python Modules

- Python is "batteries included": a rich standard library
  - Mathematics, text processing, networking, system, ...
  - A whole part of the official documentation presents them
- Third party modules are available http://pypi.org
- And can be installed very easily by the tool *pip*

The *math* module

```
$ python3
>>> factorial(5)
...
NameError: name 'factorial' is not defined
>>> from math import factorial
>>> factorial(5)
120
```

Programming in Python

Interactive mode

- Just install Python 3 and you're done!
  - On MS Windows do not forgot to ask for PATH variable update
- Open a terminal (note that in your installation Python 3 command may be either python3 or python)

```
$ python3 # or python
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
>>> 42 + 13
55
>>> price = 42
>>> price = price / 2
>>> price > 16 and price < 42
True
>>> print('Hello World!')
Hello World!
```

A first program

Create a file named `hello.py` with any text editor

```python
name = input('Your name: ')
print('Hi ' + name)
```

In a terminal, go to your file's directory

```
$ cd ~/devel/Python
$ pwd
/home/john/devel/Python
$ ls
hello.py
$ python3 hello.py
Your name: John
Hi John
```

Interactive Python

Interactive mode

- Just run `python` or `python3` without a file name as an argument
- A few other interactive interfaces: *idle3*, *bpython*, *ipython*, ...
- It is called an REPL loop:
  1. **R** *ead*: read user input
  2. **E** *val* / **E** *xecute*: evaluate expression / execute instruction
  3. **P** *rint*: print out a value if it is an expression
  4. **L** *oop*: back to step 1.
- A specific case for the *Print* step: when the *value* is *None* nothing is printed

```
>>> print('Hello')
Hello
>>> print(print('Hello'))
Hello
None
```

How to run a Python program/script?

From the command line on MS Windows or UNIX/Linux

Just give to `python3` the file name or path as an argument:

```
$ python hello.py
Hello World!
```

From the command line on UNIX/Linux

- Add the *she-bang* line on top of your file:

```python
#!/usr/bin/env python3
print('Hello World!')
```

- Allow execution permission on the file:

```
$ chmod +x hello.py
$ ls -l hello.py
-rwxr-xr-x 1 john  john ...   hello.py
$ ./hello.py
Hello World!
```

Exercise : *Ok boomer!*

A very simple first program

- Ask the user's name and age
- Call the file boomer.py

```
name = input('Your name:')
age  = int(input('Your age:'))
```

- Write greetings then « *Boomer!* » if the age is more than yours (or whatever value you want)

```
$ ./hello.py
Your name: John
Your age: 42
How are you John?
Boomer!
```


Programming best practices

Generally

- Comment your code : everything following a sharp «#» is ignored by the compiler
- Name object according to their *meaning*: *name*, *price*, *products*, and not *a*, *b*, *data*
- Read carefully compiler and runtime error messages

Specifically in Python

- Document your scripts, your fonctions, classes, modules
- Easy: write a few lines of text enclosed by three quotes or double quotes at the beginning of file or bloc of code

```
#!/usr/bin/env python3
'''This program asks for the user's name
then outputs a somewhat ironic greeting.'''
```

- It's more than a mere comment, it may be shown by a call to `help()`

Python specifics

Off–side rule

- Introducing a bloc of code (after the *if* instruction for instance) is done by a colon + a new line then indenting the lines by introducing the same amount of space at the beginning of each line of the bloc
- Use 4 spaces (no more, no less, no tabs)
- A bloc ends when indentation is back to the previous level
- *Very* peculiar way to do it...
  - { and } in C, C++, Java, C#, Perl, PHP, ...
  - (and) in LISP et Scheme, `Begin` and `End` in Pascal
  - among many others...

```
if age > 42:
    print('Boomer!')
    print('No offense.')
print('Bye ' + name + '!')
```

Python specifics

No variables (or names) declaration

- Just assign a value to them: `name = 'John'`
- Nevertheless Python is *strongly typed*: usually no automatic conversion between types

```
>>> '42' > 32
TypeError: '>' not supported between instances
of 'str' and 'int'
>>> int('42') > 32
True
```

What about non-existing names?

```
>>> product
NameError: name 'product' is not defined
```

Data and objects

Data

- *Object* stored in memory by the *Python Virtual Machine*
- Memory is allocated when needed
- Will be freed later if the object is not in use anymore (referenced), this is the *garbage collector* job
- Every object has a *type*

```
>>> 42
42
>>> type(42)
<class 'int'>
>>> type('John')
<class 'str'>
>>> id(42)      # memory address
9080320
```

Objects and names

A new name is created by an assignment

```
>>> name = 'John'
>>> age = 42 + 13
>>> type(name)
<class 'str'>
>>> type(age)
<class 'int'>
>>> id(name)
140240346121136
```

A name can be deleted

```
>>> del(age)
>>> age
NameError: name 'age' is not defined
```

Références

Every name is actually a *reference*

- Strictly speaking it is not a *variable*
- Several names can reference the *same* object

```
>>> food = 'spam'
>>> bad = food
>>> id(food)
14024034572668O
>>> id(bad)
14024034572668O
>>> food is bad
True
```

Comparing references and objects

- `is` to know if it is the *same* object in memory
- `==` to know if two objects have the same *value*

Numeric built-in types

Family of compatible types

- `int`: signed integers, unbounded
- `float`: floating point numbers
- `complex`: complex numbers
- You can freely use them alltogether in various kind of expressions:

```
>>> price = 42
>>> maximum = 49.99
>>> length = 42.0
>>> z = 1 + 2j
>>> price <= maximum        # less or equal
True
>>> price == length
True
>>> (z - 1) ** 2 == -4
True
```

Numerical operators

Arithmetic

- +, −, * (product), /, (division), // (floor division), ** (power)
- % : *modulo* (euclidean division remainder)
- ~, ^, |, &: bitwise operators
- Each of these is a call to a *dunder method*: `__add__`, `__mul__`, ...
- Complex numbers have attributes: *z.real*, *z.imag*
- More on this later, when we'll deal with classes and objects

Usual priority rules apply, you can use parenthesis to group sub-expressions.

```
>>> 1 + 2 * 3 == (1 + 2) * 3
False
```

Characters string type

Strings

- `str`: collection of characters
- Litterally expressed by enclosing text with quotes, double quotes or three of consecutive of these enclosing characters

```
>>> name = '''John'''
>>> msg  = "It's not Joe"
>>> text = 'He is a "boomer", really?'
```

- As in many other languages *backslash* has a specific meaning in strings
  - To insert the litteral inclosing char `'it\'s a "boomer"'`
  - To insert a special character: end of line `\n`, smiley `\N{grinning face with smiling eyes}`, tab `\t`
- Outside of strings a *blackslash* allow you to break lines without breaking the Python off-side rule
- *chr(n)* is character of code *n*, *ord(c)* is the code of character *c*

Strings operations

Operators

- `+` : concatenate
- `*` : repeat *n* (integer) times
- `in` : look up for a substring

```
>>> 'John' + ' ' + 'Cleese'
'John Cleese'
>>> 'spam ' * 4
'spam spam spam spam '
>>> 'spam' in 'ham egg spam sausage'
True
```

Functions

- `len()` : length, works for all kind of collections
- `int()` : integer conversion (error if not possible)
- `float()` : conversion into floating point number

Operators and assignments

It is quite usual to use an operator then assign the result to the initial name

```
>>> food = 'egg '
>>> food = food + ' spam'
>>> food
'egg spam'
>>> age = 42
>>> age = age + 1
```

A shortcut: *operator=*

```
>>> food += ' spam'
>>> age += 1
>>> qty *= 2
>>> price /= 2
```

Functions

A *function* is an object that may be *called*

```
>>> len      # object
<built-in function len>
>>> type(len)   # with a type
<class 'builtin_function_or_method'>
>>> len('spam') # call: func()
4
>>> int('42')
42
```

```
>>> 'spam'(42)
TypeError: 'str' object is not callable
```

Methods

A *method* is a function inside an object's name space

It may *modify* the object or *return* another object

```
>>> food = 'spam'
>>> food.upper()     # new object
'SPAM'
>>> food.isupper()
False
>>> food         # unchanged
'spam'
>>> food.startswith('spa')
True
```

Building complex strings

Putting up together constant and variable parts

```
>>> price = 42; prod = 'spam'
>>> "Price of " + prod + ' is ' \
    + str(price - 10) + '€'
'Price of spam is 32€'
```

More convenient: `str.format` method

```
>>> 'Price of {} is {}€'.format(prod,price - 10)
```

Since Python 3.7 : even better!

```
>>> f"Price of {prod} is {price - 10}€"
```

A lot of control is possible on string formating, look for *str.format* and *« f-strings »* in the manual.

How to find out about all these functions and methods?

Read the fine manual!

- http://docs.python.org/3.7/
- You can ask for help from the interactive interpreter `help()`, `help(str)`

You can ask Python for names in a namespace!

```
>>> dir(__builtins__)  # built-in names
[ ...'dir', ... , 'int', ..., 'len', ..., 'str', ... ]
>>> dir(str) # methods on strings
[ ..., 'lower', ..., 'isupper', ..., 'upper', ... ]
>>> import math
>>> dir(math)
[ ..., cos , factorial, ..., pi, ..., sin, ... ]
>>> from math import sin,cos,pi
>>> sin(pi/4) + cos(pi/4)
1.414213562373095
```

Booleans

`bool` type

- Values `True` or `False`
- This is what operators like `==`, `!=`, `<`, `<=`, `>`, `>=`, `is`, `is not` return
- Logical expressions can be build by `or`, `and`, `not` and parenthesis

What is `if` *expression*`:` doing?

- First it converts the expression into a boolean: `bool(expression)`
- For *0*, *0.0*, `''`, *None* ; `bool(...)`is *False*, otherwise *True*
- An empty collection is False, any non-empty is True
- If the result is *(True)*, `if` executes the following bloc of code

```
price = 42
if price:
    print("It is not free...")
```

The `if` instruction

`if` allows an optional following bloc `else`

```python
if 'spam' in food:
    food += ' more spam'
else:
    food += ' spam' # spam is mandatory!
```

Several tests can be chained by `elif` *(else if)* before the optional *else* clause

```python
if 'spam' in food:
    food += ' more spam' # more spam!
elif 'egg' in food:
    food += ' spam'      # free spam!
else:
    food += ' spam spam' # double spam!
```

`if` as an expression

The syntax `if: ...: ... else:` denotes an *instruction*

You can denote an expression instead:

```
>>> price = 42
>>> 'expansive' if price > 50 else 'cheap'
'cheap'
>>> price = 57
>>> 'expansive' if price > 50 else 'cheap'
'expansive'
```

Python's expressiveness: readable, consise, powerfull!

```
msg = 'You have ordered {} product{}'.format(n,
                    's' if n > 1 else '')
```

Exercise : *English Breakfast*

Ordering food

- If some *spam* is ordered, add *good!*, else if some *ham* is ordered add *spam* and last, (if neither *spam*, nor *ham* have been ordered) add *egg*
- Use only string operators (*in*, $+$)
- Everythin following `Command` here is user input, the rest is the program output:

```
$ python3 breakfast.py
Command: ham egg
Delivered: ham egg spam
$ python3 breakfast.py
Command: ham sausage
Delivered: ham sausage spam
$ python3 breakfast.py
Command: ham spam bacon
Delivered: ham spam bacon good!
```

Now modify your script to *add* 10 times *spam* if ever some *ham* has been ordered

```
>>> ' covfefe' * 3
' covfefe covfefe covfefe'
```

Compute and output how much *spam* will be delivered:

The `count` method on strings determine how much times a sub-string is present

```
>>> 'to be or not to be'.count('to')
2
```

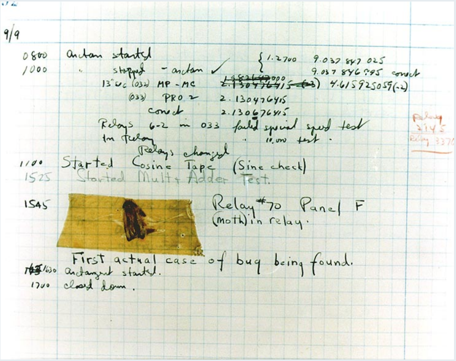
Modify your program accordingly

After the `if : elif: else:` instruction:

```
$ python3 breakfast.py
Command: ham sausage
Delivered: ham sausage spam
Spam      : 1
```

What about ordering an hamburger?

- Now try to order an hamburger
- How much spam have you got? Why?
- Oh, God, this is a *bug*...




```
>>> 'ham' in 'egg hamburger coffee'
True
```

Collections

Objects that contains multiple references to other objects

```
>>> menu
[ 'ham', 'spam', 'egg', 'sausage' ]
>>> prices
{ 'ham': 42, 'spam': 12, 'sausage': 20 }
>>> prices['spam']
12
```

A first kind of collection: *lists*

```
>>> menu = [ 'ham', 'spam', 'egg', 'sausage' ]
>>> menu[0]
'ham'
>>> len(menu)
4
```

Lists

Litteral expression for a list `[ expression, ... ]`

- Expressions enclosed by *brackets*, separated by *commas*
- Can contain any type: *str*, *int*, *list*, etc.
- Can easily be build by splitting a string

```
>>> data = [ 'a', 42, 12, -3.14, cos(pi) ]
>>> table = [ [ -2, 4 ], [ 7, 0 ] ]
>>> food = 'ham spam egg bacon spam'.split()
>>> help(str.split)
```

You can access to collections *items* with an *index*

First item is at index *0*

```
>>> data[0]
'a'
>>> table[1][0]
7
```

Extracting items and modifying lists

You can change what is the reference at a given index

```
>>> food[3] = 'sausage'
```

Negative indices start from the *end*

```
>>> food[-1]
'bacon'
>>> food[-2]
'egg'
```

You can extract sub-lists (*slices*)

```
>>> food[2:4]
[ 'spam', 'egg' ]
```

Note that we have extracted a slice from index **2** to index **4** - 1 = 3.
The item at the last index of a slice is *not included*.

Operators for lists

Operators +, * and in/not in

```
>>> food = [ 'ham', 'spam' ]
>>> food + [ 'egg', 'sausage' ]
[ 'ham', 'spam', 'egg', 'sausage' ]
>>> food * 2
[ 'ham', 'spam', 'ham', 'spam' ]
>>> 'spam' not in food
False
>>> 'cheese' in food
False
```

You can get a string back from a list of strings

```
>>> ' ; '.join(food * 2)
'ham ; spam ; ham ; spam'
```

You may be puzzled, as this is not `list.join(sep)` but `sep.join(list)`

Exercises : from strings to lists

Refactoring the previous exercise solution:

- Store the ordered food in a list of strings
- Then all tests will be made on that list instead of a unique string
- Is there much code to change?
- Is there a *count* method for lists? Is it running as expected?
- Bring back strings in the game by displaying the whole command like this:

```
**** Fawlty Towers Hotel  ****
ham
...
bacon
**** Service not included ****
```

- Is the *hamburger* bug still there? Why?

Exercise: instrospection on lists

Ask Python for all available methods on lists

Ignore at that time the ones enclosed by __

```
>>> myFood = [ 'spam', 'egg' ]
>>> dir(list)
...
>>> help(list.index)
>>> myFood.index('spam')
```

Try to guess and experiment to determine which ones are modifying the list they are called on and which one are returning *another* object (either a list or not)

What is the `sort` method doing on a list? What is it returning? Compare with the *function* `sorted`.

Modifying a list

A single item may be changed

```
>>> food = [ 'spam', 'ham', 'egg' ]
>>> food[1] = 'sausage'
>>> food
[ 'spam', 'sausage', 'egg' ]
```

Some methods modify lists too

```
>>> food.append('pudding')
>>> food.remove('spam')
>>> food.pop()
```

insert, *reverse*, *sort*, *extend*, *clear*

A list can be modified by assigning a sequence to a *slice*

A very expressive way to modify a data set

```
>>> food = 'spam ham egg sausage cheese'.split()
>>> food[1:3]
['ham', 'egg']
>>> food[1:3] = []
>>> food
['spam', 'sausage', 'cheese']
>>> food[1:2]
['sausage']
>>> food[1:2] = [ 'spam', 'pudding', 'beans' ]
>>> food
['spam', 'spam', 'pudding', 'beans', 'cheese']
```

Sequences unpacking

Allow to extract information from a *sequence* into *names*

```
>>> product = [ 'spam', 42 ]
>>> food, price = product
>>> food
'spam'
>>> price
42
```

You can use slices in order to match the number of names, and repack sub-sequences

```
>>> product = [ 'spam', 42, 'good', 10 ]
>>> food, price = product[:2]
>>> food, price, *end = product
>>> end
['good', 10]
```

Another kind of collection: Dictionaries

Dictionaries: keys and values

```
>>> pricedb = { 'spam':12, 'ham':42, 'egg':10 }
>>> pricedb['ham']
42
>>> 'ham' in pricedb # looks for keys
True
```

A dictionary can be altered

```
>>> pricedb['beans']
KeyError: 'beans'
>>> pricedb['beans'] = 7; pricedb['beans']
7
>>> pricedb.get('spam')
12
>>> pricedb.get('tomatoes',0)
0
```

Another Python collection: tuples

Tuple : immutable sequence (similar to *lists* but cannot change)

You can access to items exactly like if it were a list (indices, slices, unpacking)

```
>>> foods  = ( 'spam', 'ham', 'egg', 'beans' )
>>> prices = ( 12, 42, 10, 7 )
>>> foods[2]
'egg'
>>> foods[1:3]
( 'ham', 'egg' )
>>> prices[1] = 44
TypeError: 'tuple' object does not support item
assignment
>>> prices_list = list(prices)
>>> prices_list[1] = 44; prices_list
[ 12, 44, 10, 7 ]
```

Another kind of Python collections: sets

set and *frozenset* : mutable an imutable sets

```
>>> food = { 'spam', 'ham', 'egg', 'ham' }
>>> len(food)
3
>>> food
{ 'spam', 'ham', 'egg' }
```

Useful (but non only) to remove duplicates

```
>>> food = [ 'spam', 'ham', 'egg', 'ham', 'spam' ]
>>> food = list(set(food))
>>> food
['ham', 'spam', 'egg']
```

Immutable and mutable

Among all built-in types we have been talking about, which ones are mutable or immutable?

Immutable

- Numbers : *int*, *float*, *complex*
- Strings : *str*
- Tuples : *tuple*
- Frozen Sets : *frozenset*
- Booleans *bool*, *NoneType*

Mutable

- Lists : *list*
- Dictionaries : *dict*
  - But *keys must* be of an immutable type
- Sets : *set*

Types conversion

We can convert across most collections

```
>>> int('42')
42
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple([ 1, 2, 3 ])
(1, 2, 3)
>>> list( (1,2,3) )
[1, 2, 3]
>>> ''.join([ 's', 'p', 'a', 'm'])
'spam'
```

All types names *(str, int, list, . . .)* are *functions*

- Try to do their best to convert into the specified type, may fail with error
- Without argument returns *zero*, *void*, *false*, *nothing*, . . .

`for` loop statement (instruction)

`for` allow to walk through a collection

```
>>> foods = 'spam egg ham'.split()
>>> for food in foods:
        print(food)
spam
egg
ham
```

Works for all collections (more generally any *iterable*)

- Lists, including slices
- Tuples, including slices
- `range(n,m,p)` : integers from *n* to *(m-1)* with step *p*
- Dictionaries (goes through the *keys*)
- Sets
- and also *(iterables)* : files, database queries, CSV file readers, ...

Loops and unpacking

Items can be collections (lists, tuples, strings)

```
>>> foods = [ ('spam', 12), ('ham', 42) ]
>>> for elt in foods:
        print(elt)              # a sequence (tuple)
        food, price = elt       # items
        print(food,price)
('spam', 12)
spam 12
('ham', 42)
ham 42
```

for can *walk through* items **and** *unpack* them

All in one go!

```
>>> foods = [ ('spam', 12), ('ham', 42) ]
>>> for food, price in foods:
        print(food,price)
spam 12
ham 42
```

Especially convenient for dictionaries!

```
>>> prices = { 'spam':12, 'ham':42 }
>>> for food, price in prices.items():
        print(food,price)
spam 12
ham 42
```

Reading a simple text file

Let's read a simple file `digits.txt` the same way!

```
    one     1
    two     2
    ...
    nine    9
```

A *for* loop iterates through the file

```python
for line in open('digits.txt'):
    print(line.rstrip()) # remove trailing line feeds,
                         # anyway print adds his own
```

Output

```
    one     1
    two     2
    ...
```

Doing better

Using *with* statement and processing data

with is a statement making sure that the bloc is not executed in the case of failure in opening file(s) and it will close them if needed.

```
with open('digits.txt') as input:
    for line in input:
        word, value = line.rstrip().split()
        value = int(value)
        print('{} is called {} ' \
              'in English'.format(value,word))
```

Don't reinvent the wheel

It is even simplier for a CSV file or a database: for a CSV file, csv module takes care of spliting lines, a database DB connector even takes care of datatypes.

Exercice : *Menu à la carte*

A new script `alacarte.py` built on the previous one

Again ask the user to order various foods and store all of them in a list.

- Display all of them with a *for* loop
- Assign an empty list to a name `up_foods`
- In another *for* loop going through the list of ordered foods add each of them, changed to uppercase, in `up_foods`
- Display `up_foods` items

The bill...

Create a file `prices.txt` like this:

```
ham     42
spam    12
beans   8
sausage 11
cheese  10
...
```

The bill (continued)

The bill!

- Read that file and store the relevant information into a dictionary (you can start from an empty dictionary {})

```
with open(...) as ...:
    pricedb = {}
    for line in ...:
        prod, price = line.rstrip().split()
        price = float(price)
        pricedb[prod] = price
print(price)
```

- Display the bill including every item, its price and the total amount to be paid for breakfast.

Exercise : Cryptography

Build the alphabet

- In a loop make `i` vary from the code of ‘a’ to the code of ‘z’

  ```
  range(ord('a'), ord('z') + 1)
  ```
- At first display `chr(i)`
- Modify the loop to append chr(`i`) to a list:

  ```
  alphabet = []
  for i in range(ord('a'), ord('z') + 1):
      # add chr(i) to alphabet
      ...
  print(alphabet)
  ```

Let’s change the letters

Given a latin ASCII letter with code k - What is this doing?

```
secret = 5
chr( ( k - ord('a') + secret ) % 26 + ord('a') )
```

Exercise : Cryptography (continued)

This is what Julius Cesar did during Gallic Wars!

- Ask the user for a whole sentence
- Loop the string, for all letter you will display another letter:
  `chr( (ord(c) - ord('a') + 5) % 26 + ord('a'))`
- Display these characters with: `print(..., end = '')`

```
for c in text.lower():
    if c in alphabet:
        print(..., end = '')
    else:
        print(c)
print()
```

- Building `alphabet` was useless, we reinvented the wheel:

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Another loop statement: `while`

Another instruction: `while`

Execute a bloc *while* a given condition is true:

```
>>> ans, menu = '', []
>>> while ans != 'spam': # ask for spam to end
...     ans = input('Food: ')
...     menu.append(ans)
...
Food: ham
Food: egg
Food: spam
>>> print(menu)
[ 'ham', 'egg', 'spam' ]
```

Control of execution flow

Escaping from *for* and *while* loops: *break*

- When you have encountered a anomaly, found what you were looking for, an user asks to quit
- With an optional *else* statement you can execute code *only if* the loop exited "normally" (no break)

```
while True:
    ans = input('Name (!END to quit) : ')
    if ans == '!END':
        break
```

You can jump to the next iteration with *continue*

```
for line in input:
    if line.startswith('#'):
        continue
    data = line.strip().split()
    .... # Process data in line
```

Comprehensions

A *expression* build with `for`

Building a list from another one with a pure functional expression.

```
>>> foods = [ 'spam', 'ham', 'egg', 'beans' ]
>>> [ food.upper() for food in foods ]
['SPAM', 'HAM', 'EGG', 'BEANS']
```

You can filter out items

```
>>> [ food.upper() for food in foods \
                    if food != 'egg' ]
```

And use *if* as an expression too

... *if* test *else* ... we've seen before

```
>>> [ food.upper() if food != 'spam' else 'beuh' \
                   for food in foods \
                    if food != 'egg' ]
```

Other datatypes

Various modules extend built-in datatypes

- *collections* provides *defaultdict, namedtuple, deque, Counter*
- *enum* provides *Enum*
- *array* provides arrays of homegeneous data
- Examples at:
  https://framagit.org/jpython/miscellaneous-python

```
text = 'to be or not to be'.split()
d = defaultdict(int)
for w in text:
    d[w] += 1
for w,n in d.items():
    print('{}: {} times'.format(w,n))
print(Counter(text)) # no need for loop...
```

Loops under the hood: *iterators*

The iteration protocol

- All objects that obey the *iteration protocol* can be used in *for* loops and are sequence-like objects
- All collections follows the iteration protocol

```
>>> it = iter(['spam','ham'])
>>> next(it)
'spam'
>>> next(it)
'ham'
>>> next(it)
StopIteration
```

They are everywhere

- Opened files, CSV readers, DB requests, ...
- Views on dictionaries (items, values, keys)
- *zip*, *enumerate*, ...

Iterators

Building iterators

- *itertools* module provides more ways to build iterators
- *chain*, *product*, *repeat*
- Comprehension iterators:

```
squares = ( x**2 for x in range(10) )
for elt in squares:
    print(elt)
```

More on this later

- Can be build by functions using the *yield* instruction
- Can be build by objects implementing specific dunder methods `__iter__`, `__next__` and raising `StopIteration` if needed

Functions

A function allows to name and reuse code

Defined by the `def` instruction

```
>>> def printLicense():
        print('Available under the terms')
        print('of the WTFPL license')
        print('more at: http://www.wtfpl.net/')
>>> printLicense()
Available under the terms
of ...
```

If the file `wtfpl.py` defines this function you can use it as a library module yet!

```
>>> from wtfpl import printLicense
>>> printLicense()
```

Arguments and return values

A function may receive arguments

```
>>> def cry_if_spam(string):
        if 'spam' in string:
            print("I do not like SPAM!!!")
>>> cry_if_spam('ham egg')
>>> cry_if_spam('ham spam egg')
I do not like SPAM!!!
```

A function can return something (otherwise it is *None*)

```
>>> def where_is_spam(menu):
        return menu.index('spam')
>>> where_is_spam(['egg', 'ham', 'spam', 'beans'])
2
```

Optional arguments

Just provide a default value

```
>>> def price_of(food, discount = 0):
        prices = { 'spam': 12, 'ham': 42 }
        price  = prices.get(food, 0)
        price  *= (100 - discount)/100
        return price
>>> price('ham',10)
37.8
>>> price('ham')
42
```

Calling functions

Arguments can be passed by *position*

```
>>> price_of('food',10)
```

Or by *keywords*

```
>>> price_of(discount = 20, food = 'spam')
```

Or both... like when using the *print* function

```
>>> print('spam', 'ham', 'egg', 42, sep='\n')
```

Variable positional arguments

You can define a function accepting an unknown numbers of positional arguments

```
def sum_of_squares(*args):
    return sum( [ elt ** 2 for elt in args ] )

print(sum_of_squares(1,4,42,12))
```

You can unpack a sequence as arguments

```
foods = [ 'spam', 'ham', 'egg' ]
print(*foods)
```

Collecting and unpacking keywords arguments

You can collect keyword arguments in a dictionary

```
def insert_into_db(**kwargs):
    create_user(name=kwargs['name'])
    set_food(kwargs[name],food=kwargs['food]')

insert_into_db(name='Joe', food='spam')
```

You can unpack a dictionary as keyword arguments

```
user_info = { 'name':'John', 'food':'spam' }

display_form(**user_info)
```


Names and scopes

Whenever a name is *dereferenced*

- First a lookup is done *locally* (at function level)
- Then in all *enclosing* functions (if any nested `def`)
- Then *globally* (at script or module level)
- And last at *built-in* level
- This is the **LEGB** rule

When you *assign* a name

- Instruction = or *operator*= (not all the time for the latest)
- Only *local* scope is altered
- You can override this, this is a *bad* idea

Modifying arguments

- As every name is a reference, arguments are received as such
- If an argument is *mutable* and is *modified* – say *data.append(...)* – it will be seen from “outside”

Generator functions

Another way to build iterators

- Use *yield* instruction to release value through *next()* calls
- *return None* to raise *StopIteration*
- Lazy evaluation

```
def genSq(start=0, end=10):
    while True:
        if start < n:
            yield start**2
            start += 1
        else:
            return None

for i in genSq():
    print(i)
```

Lambda anonymous functions

Building a function without a name

```
>>> (lambda x,y: x**2 + y**2)(2,3)
13
>>> f = (lambda n: sum(range(n)))
>>> f(42)
861
```

LISP-ish construction related to Church *lambda-calculus*

- Useful for a quick definition
- You can put functions in collections and evaluate them later
- To pass a one-shot function as an argument

```
>>> area['rectangle'](2,3) + area['triangle'](2,3)
9.0
>>> from functools import reduce
>>> reduce( (lambda x,y: x*y), range(1,10))
362880
```

Exercice : *Professional Breakfast*

From a previous exercise solution

We can read and store information about various foods from a file.

1. Write a function returning the price of a food
2. Write a function accepting a list of ingredients and returns the total prices if one orders all of them
3. Test this function with a list of foods
4. Rewrite the function to accept directly all foods as arguments instead of a single list. Call it by passing the list of foods unpacked.

Debugging

Read and interpret error messages

- Python is raising the error as close as possible to the *real* issue
- Interpret exceptions:
  - *is not callable* : this is *not* a function
  - *has no attribute...* : typo in method, bad type or unsupported operator

Add debugging code

- *print()*, *sys.stderr.write()*
- *logging* module

Step by step debugging

- `pdb` from the command line
- *pdb* can be driven by an IDE like *PyCharm* or *Eclipse* with *PyDev* extension, *Spyder*

Unit test

Various tools and modules

- *unittest* from the standard library
- *pytest* can be installed easily
- *nose* is a very popular fork of *pytest*

Example with *nose*

```
from yourmodule import price_of
def test_price_of_food():
    p = price_of('ham')
    assert p == 42
    p = price_of('spam')
    assert p == 12
    p = price_of('notfood')
    assert p == 0
    p = price_of('ham',10)
    assert p == 42 - 42/10
```

Running tests with *nose*

Just run *nosetests*

```
$ nosetests test_priceof.py
....
------------
Ran 4 test in 0.001s
OK
```

You can even get a test coverage report

```
$ nosetests --with-coverage \
            --cover-html    \
            test_priceof.py
$ firefox cover/index.html &
```

Exercise: Spy against Julius Cesar

Context

In *Gallic Wars* Julius Cesar explained the cryptographic system he used: a circular permutation on the latin alphabet.

The story

We have intercepted an encrypted message from the Romans. We happen to know that it contains the word *spam*. We have to break that code!

Here is the message:

```
Z UF EFK CZBV JGRD RK RCC!!! Z UF EFK NREK KYRK!
A'VJGVIV HLV MFLJ MFLJ VKVJ RDLJVJ GVEUREK TV TFLIJ...
```

Break the code!

Steps for breaking the code

Encode char and string

- Write a function `encode_char(c,n)` returning the encrypted result for char *c* if it is a letter, *c* otherwise (help yourself with the *string* module)
- Write a function `encode_string(msg,n)` returning the encrypted version of string *msg*
- Write a `decode_string` function

Brute force

- In a loop try to decode the secret message with key *k* with *k* looping over *range(1,26)*
- Break if the hint ('spam') has been food and display the decyphered message
- An *else* clause at the end of the loop allows you to handle failure (or you can use a boolean *flag*)

What is Object Oriented Programming?

An object is a container

- It has *attributes*: all are Python objects
- Some are *data*: lists, integers, strings, . . .
- Others are *methods*: functions acting on/using the object

A class is a template for objects

- A given object is an instance of a class
- Class are basically the same thing as types in Python

An instance can be built by calling the constructor

- In Python a special method called `__init__`
- Strictly speaking this is an *initializer*
- As all methods it will receive *self* as a first argument
- Usually populates attributes from arguments

Example: the built-in complex class

Built-in types usually have a litteral syntax

```
>>> z = 1 + 2j # instance of complex
>>> type(z)    # its class
<class 'complex'>
>>> z.real     # instance data attribute
1.0
>>> z.conjugate() # a method call
(1-2j)
>>> dir(z) # all instance attributes
```

They are still objects as any others

```
>>> z = complex(1,2)
>>> z + (1 + 3j)
(2+5j)
>>> z.__add__(1 + 3j)
(2+5j)
```

Object Oriented Programming

Python is object oriented from the ground up

- Every type is a class
- Everything is an object

You can create your own classes

- Specify in `__init__` method what to do at initialisation time
- Will be called when creating an *instance* of that class

```
class Point:
    # Called implicetely by Point(x,y,label)
    def __init__(self,x,y,label):
        self.x = x
        self.y = y
        self.label = label
    def show(self):
        print('{}({}, {})'.format(label,x,y))
```

Normal and *dunder* methods

Normal methods are called by their names

```
p = Point(3, 4, 'center')
p.show() # same as Point.show(p)
```

Specials methods called *implicitely*

- `__init__` at initialisation stage
- `__add__`, `__sub__`, `__mul__`, ... for +, −, *, ... operators
- `__str__` when converted into string
- `__repr__` when displayed in REPL

```
def __str__(self):
    return '{}({}, {})'.format(label,x,y)
def __repr__(self):
    return 'Point({}, {}, {})'.format(x,y,label)
```

Specials methods and operators

How is an expression with an operator evaluated?

When evaluating $a + b$...

- First `a.__add__(b)` is tried
  - Which is actually (class of a).`__add__(a,b)`
- Then `b.__radd__(a)` is tried
  - Which is actually (class of b).`__radd__(b,a)`

It's not a big deal to overload operators...

- Just define `__add__`, `__sub__`, `__mul__`, ...
- Don't shoot yourself in the foot though!

Inheritance

A class can inherit from another one

- The subclass can redefine and add methods and attributes
- Either from built-in, modules or custom classes
- Multiple inheritance is supported

```python
from enum import Enum
from operator import add
class Move(Enum): # Inherit from Enum
    UP    = ( 0 , -1 ) # No need for __init__
    DOWN  = ( 0 ,  1 ) # Just provides allowed
    RIGHT = ( 1 ,  0 ) # constant values
    LEFT  = (-1 ,  0 ) # as class attributes
    # called when evaluating coord + self
    def __radd__(self,coord):
        return tuple( add(*t) for t in
                      zip( self.value, coord ) )
print( (3,4) + Move.UP )
```

Modules

Any Python script is already a module

Just import `mymodule.py` the way you want:

```
import mymodule
mymodule.myfunc(42)
from mymodule import myfunc
myfunc(42)
import mymodule as mm
mm.func(42)
```

Conditional execution

You can execute code only if executed, *i.e.* ***not*** *imported*

```
if __name__ == '__main__':
    # Test code, not executed if imported
```

Exercise: Objects and inheritance

Moving a tuple of coordinates the other way around

- Instead of inheriting from *Enum* for a move, create a class *Coord* inheriting from *tuple*
- Define the `__add__` method which is called when a tuple is added to it (instead of defining `__radd__` in the Move type)

```
>>> c = Coord(3, 4)
>>> print( c + Move.UP )
(3, 3)
```

- Define the `__str__` method:

```
>>> print(c + Move.UP)
Coord(3, 3)
```

- You can also do sanity checks

```
if not isinstance(move.value, tuple) \
        and len(move.value) == 2:
    raise ValueError('Not a 2-valued tuple')
```

More on objects

Things to know about Python classes and objects

- Parent methods can be called in overrided methods

```
def __init__(self, ...):
    # my stuff ...
    super().__init__(...) # no self
def mymethod(self, ...):
    # my stuff ...
    super().mymethod(...) # no self
```

- There are no private attributes or methods
  - Mark internal attributes with a prefix _ or __
  - Then then won't appear in *help(YourClass)*
- Class names usually should have a leading capital (PEP 8)
- You can define *class methods* and *static methods*
- You can create *accessors* (i.e. calling functions implicitely when dereferencing/assigning a name)
- Most of this is based on *decorators*

Installing modules

Modules available on PyPI

- All modules on pypi.org are installable with `pip`
- Especially convenient in a virtual environment

```
$ python3 -m venv venv
$ source venv/bin/activate
# MS Windows: venv\Script\activate.bas
(venv) $ pip install requests
(venv) $ python3
>>> import requests
```

Anaconda

- Anaconda suite provides a similar tool: `conda`

```
(base) $ conda install ...
```

Anaconda

Anaconda is a single package installing Python and a huge set of modules and tools

- *IPython* : improved REPL command line interface
- *numpy* : efficient number and multi-dimensionnal arrays computation
- *pandas* : data indexing, querying and aggregation
- Both provide objects with similar interfaces (*iteration protocol*, method names) to built-in Python or array module objects
- Interface nicely with CSV files, SQL databases, even Excel files
- *scipy* : numerical analysis, linear algebra, statistics
- *scikit-learn* : classification, clustering, regression
- *matplotlib*, *seaborn* : data visualization, imaging

Note that all these package may also be installed *independantly* from Anaconda by *pip*.

Anaconda

Graphical User Interface

- *anaconda-navigator* gives access to most graphical tools
- *Jupyter* : notebook oriented Web interfaces
- *Spyder* : Integrated Development Environment
- PyCharm can also interfaces itself with Anaconda

More packages (set of modules) can be installed by `conda` command.

Notebooks

- You can share notebooks IPython files saved by Jupyter
- Many are available on the Internet

Numpy

Efficient numerical and array types

```
>>> import numpy as np
>>> np.array( [ [1,2,3] , [4,5,6] ] )
array([[1, 2, 3],
       [4, 5, 6]])
>>> t = np.array( [ [1, 2, 3] , [4, 5, 6] ] )
>>> type(t[0][1])
<class 'numpy.int64'>
>>> t = np.array( [ [1, 2, 3] , [4, 5.0, 6] ] )
>>> type(t[0][1])
<class 'numpy.float64'>
```

Mathematical functions

```
>>> np.exp(t)
array([[  2.71828183,   7.3890561 ,  20.08553692],
       [ 54.59815003, 148.4131591 , 403.42879349]])
```

Arrays in numpy

Arrays creation

- Can be generated from sequences, constant or random values, ranges, files
- Can be reshaped, iterated through, flattened
- Items can be addressed by single or multiple index, slices

Operations on arrays

- Functions acting on elements
- Linear algebra (transpose, multiply)
- Compound functions (sum, mean, average, . . .)

```
>>> np.sum(t)
21.0
>>> np.sum(t, axis=0)
array([5., 7., 9.])
>>> np.sum(t, axis=1)
array([ 6., 15.])
```

Pandas series

Build on top of Numpy

- Dictionary-like indexing
- Both sequence-like and dictionary-like objects

```
>>> data = pd.Series([42.42, 3.14, 1.0],
                     index=['price','size','weight'])
>>> data['price']
42.42
>>> data
price     42.42
size       3.14
weight     1.00
>>> data['size'] == data[1]
True
```

Pandas series

Database-like object

```
>>> data.where( (lambda x: x > 2) )
price      42.42
size        3.14
weight       NaN
dtype: float64
```

Filtering out NaNs

```
>>> data.where( (lambda x: x > 2) ).dropna()
price     42.42
size       3.14
dtype: float64
```

Pandas dataframes

Both are build on top of np.arrays

- Series are general sequence and dictionary-like one dimensional objects
- Dataframes are general sequence and dictionary-like two-dimensional objects

```
>>> prices = pd.Series( { 'spam':12.5, 'egg':4.3 } )
>>> qty    = pd.Series( { 'spam':4, 'egg':12 } )
>>> stock  = pd.DataFrame( { 'price': prices,
                              'qty': qty } )
>>> stock
        price  qty
  spam   12.5    4
  egg     4.3   12
```

Dataframe manipulation

You can add columns like you add dictionary entries

```
>>> stock['value'] = stock['price'] * stock['qty']
>>> stock
      price  qty  value
spam   12.5    4   50.0
egg     4.3   12   51.6
```

And aggregates values

```
>>> stock.sum()
price     16.8
qty       16.0
value    101.6
```

Data filtering

```
>>> stock[stock.price > 5]
      price  qty  value
spam   12.5    4   50.0
```

How can it works??

- Isn't `stock.price > 5` supposed to be a boolean?
- No it isn't! Not at all!

```
>>> stock.price > 5
spam      True
egg      False
Name: price, dtype: bool
>>> type(stock.price > 5)
<class 'pandas.core.series.Series'>
```

Pandas *massively* overloads comparison operators.

Numpy, Pandas, Matplotlib, and data science ressources

- Python Data Science : book and Jupyter notebooks
  https://jakevdp.github.io/PythonDataScienceHandbook/
- Scipy/Numpy Introduction
  https://sites.engineering.ucsb.edu/~shell/che210d/numpy.pdf
- Scikit tutorial
  https://scikit-learn.org/stable/tutorial/index.html
- Matplotlib Pyplot tutorial
  https://matplotlib.org/tutorials/introductory/pyplot.html