Intermediate Python -- Part 1

Thierry Géraud

2024

```python
print("Hello world!")
```

Test: write a Python program that input a number of lines and stores the strings in an appropriate structure

such as:

```
 *
***
```

or:

```
  *
 ***
*****
```

email me your solution at thierry.geraud@epita.fr with the tag [INTPY] in the subject

Let's start with a few questions...

What language(s) do you know?

```
your answers : C/C++, ruby, javascript, HTML, brainfuck, ... but not "english, or french..."
```

1. What is Python?
2. How do you use it?
3. What for?
4. Describe Python...

```
answers...
```

We will come back to this in a few minutes...

`$e^{i\pi} = 1$` gives: $e^{i\pi} = 1$

What have we here?

```
answer: a description language (LaTeX)... not a programming language
```

```
% for i in ls -d *; echo $i
```

What is it?

Is it programming?

```
answers:
- a loop in shell (zsh); that is an instruction; we are scripting
- we do not have a program at the end, so no
```

What is a *programming* language?

```
answer:
```

read https://en.wikipedia.org/wiki/Programming_language

Courses / lectures + sessions of practical work

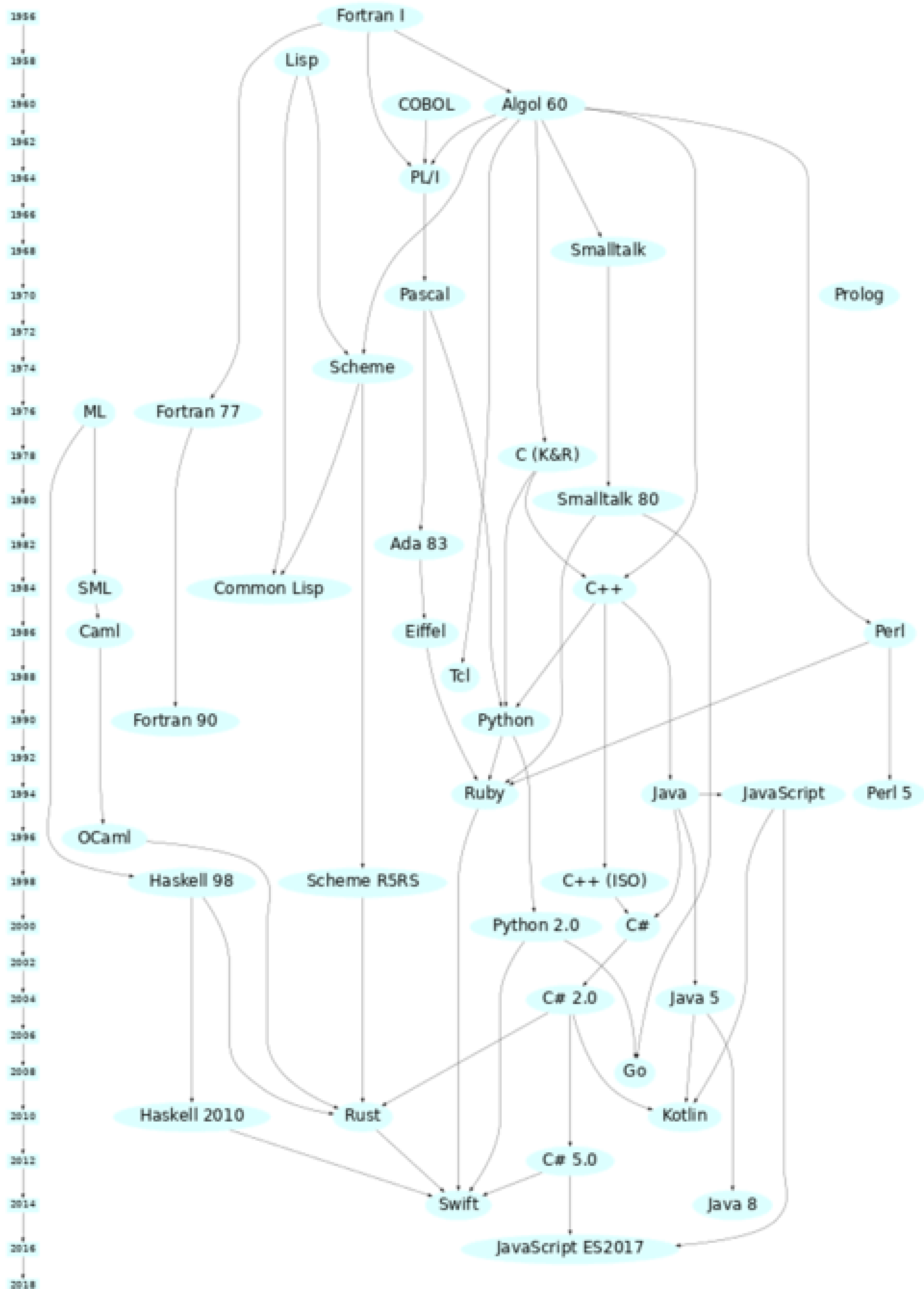
Your 1st project: display a map described by a file

Characterization of Python

Python is:

- free and open-source software
- portable
- a programming language which is
  - *high-level*
  - *general-purpose*
  - *multi-paradigm*
- easy to learn, easy to code, easy to read → accessible
- equipped with a large standard library, plus a vast range of libraries.

history of programming languages → http://tinyurl.com/IntPyLanguages



Fetch "<lang> → Python"

High-level language

compare with some *assembler* code:

```
Fib PROC
   mov eax, 1
   xor ebx, ebx
   xor edx, edx
L1:
   add eax, ebx ; eax += ebx
   mov ebx, edx
   mov edx, eax
loop L1
   ret
Fib ENDP
```

General-purpose language

- broadly applicable across *application domains*
  - e.g., bank, medicine, science
- lacking specialized features for a particular domain
  - a counterexample: solve some constraint-based problems (use Prolog instead)
  - note that libraries (except the standard one(s)) are not part of the language
  - a related key idea: a language now comes with an environment, and a community

A multi-paradigm language

Python is:

- imperative
- procedural
- structured
- object-oriented
- somehow a bit
    - functional
    - generic

Imperative

```
a = 0
a += 1
```

a series of instructions that change the state of the running program

Procedural

```
def fact(n):
    return 1 if (n == 0 or n == 1) else n * fact(n - 1)
    # an example of "literate programming" here
```

a procedure is great to factor code, to be called many times

Structured

- with control structures (e.g., for, if)
- with blocks -- thanks to indentation

```
for i in range(len(lst)):
    if i % 2 == 0:
        print(lst[i], end = '\n')
        i = i - 1
    elif i == 7:
        break
```

- subroutines

```
print(lst)  # this is a procedure call
```

subroutines are procedures (functions) and methods

compare with *basic* code:

```
05 HOME : TEXT : REM Fibonacci numbers
10 LET MAX = 5000
20 LET X = 1 : LET Y = 1
30 IF (X > MAX) GOTO 100
40 PRINT X
50 X = X + Y
60 IF (Y > MAX) GOTO 100
70 PRINT Y
80 Y = X + Y
90 GOTO 30
100 END
```

and

```
GOSUB
```

for subroutines

Object-oriented

```
class Door:
    def __init__(self, number, status='closed'):
        self.__number = number
        self.status = status
    def open(self):   # this is a method
        self.status = 'opened'
    # cut

d = Door(42)
d.open()             # this is a method call
print(d.status)
```

in this code snippet, 'd' is a Door

A bit functional

```python
l = list(range(0, 4))
l2 = list(filter(lambda x: x > 2, l))
```

we have the function: $x \mapsto x > 2$ (this is not a procedure)

A bit generic

```python
T = TypeVar('T')
def first(seq: Sequence[T]) -> T:
    return seq[0]
```

returns the first element

About types / typing

Object = instance of a type / result of the *instanciation* of a type

Type = *description* of all the objects with this type

```
d0 = Door(0)
dd = [Door(i) for i in range(10)]
```

- here d0 and dd[i] are doors / are of type Door
- they are independant, yet they behave the same way

A simple test than you shall pass

run

```
print(type(d))
print(first(d))
```

on

```
d = (0, 1, 2)
d = [0, 1, 2]
d = {0, 1, 2}
d = {'pi': 3.14, 'e': 2.72}
```

what are the output? why?

Python has some built-in types:

- Numeric data types: int, float, complex
- String data types: str
- Sequence types: list, tuple, range
- Binary types: bytes, bytearray, memoryview
- Mapping data type: dict
- Boolean type: bool
- Set data types: set, frozenset

multiple-item data types are called *collections* or *containers*

Comment the difference(s):

```
T = TypeVar('T')
def first(seq: Sequence[T]) -> T:
    return seq[0]
# versus
def first(seq):
    return seq[0]

def sqr(x: int) -> int:
    return x * x
#versus
def sqr(x):
    return x * x

b: int = 2
#versus
b = 2
```

```python
b: str = 2
print(type(b))

def doit(i: int):
    print('int', i)

def doit(s: str):
    print(type(s), s)

doit(0)
doit("0")
```

What happens? Why?

use a linter!

```python
def foo(i: int):
    pass

foo('toto')

```

and a type checker:

- Mypy (by Dropbox), Pytype (by Google), Pyright (by Microsoft)
- Pyre/Pysa (at Facebook and Instagram)

```
theo@tsee Cours Intermediate Python % mypy test.py

test.py:4: error: Argument 1 to "foo" has incompatible type "str"; expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Variables, values, and types

py:

```
a = 1                     # implicit type for 'a' -- typed by the compiler
b: int = 2                # explicit type for 'b'
b = "a string"            # do compile; 'b' is now a string
```

Pascal:

```
var b : integer = 2;  // types have to be explicit
```

modern C++:

```
auto a = 1;               // implicit type
int b = 2;                // explicit type
auto b = "a string";  // do not compile: 'b' is already defined!
```

py:

```python
a = 1
ida = id(a)
a = 2
print(id(a) == ida)  # gives: False
```

we cannot change the value of the integer whose identity is 'ida'

⇝ integers are immutable in Python

py:

```
a = 1
ida = id(a)
b = a  # 'b' is actually a reference to 'a'; we can say that 'b' is 'a'
print(id(b) == ida)  # gives: True

b = 2  # now 'b' designates a new integer
print(id(b) == ida)  # gives: False

print(a)  # gives 1
```

- single-item data types (integers, floats, complex numbers, Booleans) are immutable
- strings and tuples are immutable

whereas

- lists, sets, and dictionaries are mutable

Exercice

```python
def foo(b):
    print(id(b), b)

def bar(c):
    print(id(c), c)
    c = 2
    print(id(c), c)

a = 0
print(id(a), a)
foo(a)
bar(a)
print(id(a), a)
```

What is printed? Explain.

What the rationale behind it?

py:

```python
l = [1, 'a string']
idl = id(l)
l.append(Door(0))
print(id(l) == idl)  # gives: True, lists are mutable

print(l)  # gives: [1, 'a string', <__main__.Door object at 0x7f568add0790>]

for e in l:
    print(e)  # gives: ???
```

Explain:

- why the behavior hopefully differs from the previous example
- the output of print(l)
- the output of the for loop, and how we can make it possible?

```python
class Door:
    def set_number(self, number):
        self.__number = number
    # cut
```

py:

```python
# cont'd
print(id(l), id(l[2]), l[2])
for e in l:
    if isinstance(e, Door):  # same as: if type(e) == Door:
        e.set_number(2)
print(id(l), id(l[2]), l[2])
```

what is the output?

what have we done here?

Structured programming and blocks

Python:

```python
def doit(lst):
    for i in range(len(lst)):
        if i % 2 == 0:
            print(lst[i], end = '\n')
            i = i - 1
        elif i == 7:
            break
```

blocks start with ':' and rely on indentation

Equivalent code in C++:

```
void doit(std::list<int>& lst)
{
    for (int i = 0; i < lst.size(); ++i)
        if (i % 2 == 0) {
            std::cout << lst[i] << '\n'; // do not compile!  Why?
            i = i - 1;
        }
        else if (i == 7) {
            break; }  // the two braces are useless here
}
```

blocks are delimited by '{' and '}', and indentation does not matter

an instruction (not a block of instructions) ends with ';'

in Python indentation matters

```
i = 1
 i = 0  # gives: IndentationError: unexpected indent
```

and it is the key to see blocks

```
i = 0
while i < 5:
    print(i)
    i += 1
```

vs

```
i = 0
while i < 5:
    print(i)
i += 1
```

compare with *COBOL* (1959):

column 7 is for an indicator:

- '*' means that it is a comment
- '-' means that it is the continuation of the previous line
- '/' means that, when printing your code listing, it would page break
- 'D' means that the line would only compile when in debugging mode

compare with *FORTRAN 66*:

```
      program circle
      real r, area

c This program reads a real number r and prints
c the area of a circle with radius r.

      write (*,*) 'Give radius r:'
      read  (*,*) r
      area = 3.14159*r*r
      write (*,*) 'Area = ', area

      stop
      end
```

Memory

```
a = 1  # new object => memory allocation
b = a  # no new object
b = 2  # b is a new object => memory allocation
       # no need for explicit memory deallocation
```

py:

```
d = Door(1)          # new object
print(d.status)
d.open()             # method call
                     # no need for memory deallocation
```

Python relies on a *garbage collector* to free memory

Read about

- heap and stack
- reference counting
- the mark and sweep algorithm
  https://en.wikipedia.org/wiki/Memory_management
  https://en.wikipedia.org/wiki/Tracing_garbage_collection

The programmer can not care about memory; yet he/she shall know:

```
del my_large_container
gc.collect()
```

Python as an object-oriented language

A *particular* way of thinking:

a program = data types and algorithms (functions)

example in C:

```
struct rectangle {                          // a data type
    float width, height;
};

void scale(rectangle* r, float s) {  // an algorithm
    assert(r != NULL and s > 0);
    r->width *= s;
    r->height *= s;
}
```

Compare:

```
dta = [("EPITA", "Kremlin-Bicetre"), ("Sorbonne University", "Paris 5")]

print("name=", dta[0][0], "where=", dta[0][1])
```

with:

```
@dataclass
class University:                   # what have we here?
    name: str = None
    location: str = None

dta = [University("EPITA", "Kremlin-Bicetre"),
       University("Sorbonne University", "Paris 5")]

print(f"name = {dta[0].name} where = {dta[0].location}")

print(dta[0])  # works directly!
```

```python
from dataclasses import dataclass

@dataclass
class University:                    # a data type
    name: str = None                 # grouping two strings
    location: str = None
```

meaning that a university is composed of two data: name and location, both being strings

```python
u1 = University("EPITA", "Kremlin-Bicetre")   # a particular university, u1

print(u1.name)  # gives: EPITA -> so *explicitly* print its name

print("Paris" in u1.location)   # more readable than 'u1[1]'
```

we have defined a type:

- to be able to have objects with that particular type
    - a "University" is far more precise than "just two strings"
- to store data and explictly access to one piece of data
    - ".location" is way more explicit than "[1]"

that is what you have in all python libraries: **types**

- they are the keystones of these libraries
- the librairies also provide **algorithms**
    - doing stuff / transforming data is the key features of these libraries

```python
# a data type

@dataclass
class University:
    name: str = None
    location: str = None

# an algorithm, dedicated on objects being a University

def relocalize(u: University):
    if u.location == "Kremlin-Bicetre":
        u.location = "Grand Paris"
```

sample use:

```
dta = [University("EPITA", "Kremlin-Bicetre"),
       University("Sorbonne University", "Paris 5")]
# 'dta' is a list of two universities

for u in dta:
    relocalize(u)
# we have relocalized all universities in 'dta'
```

in:

```python
# a data type
@dataclass
class University:
    name: str = None
    location: str = None
```

we say that:

we have define a **class** (a type) with two **attributes** (named 'name' and 'location')

in this example, its is a particular class that has only data; it is "just a data type"

in this example, algorithms are defined *on the side* (usually nearby)

- this is the case of 'relocalize'
- algorithms are *not within the type*

this example corresponds to:

a program = data types and algorithms (functions)

Another way to describe programs:

- **a program = a set of types and their relationships**
  - so having a a type is having both data *and* algorithms
  - and the next level will be to think about their relationships...

this is the *object-oriented* way, and Python is an OO language

we are moving from scripting (toy use of Python) to programming (industrial use of Python)...

now:

a type = data + algorithms on these data

in object-orientation, a class has *attributes* and *methods*:

description-level	instance-level	meaning
class	object	an entity (*)
attributes	data	its state
methods (*properties*)	(sub)routines	its behavior

(*) a thing that has its own identity and whose identity conforms to what it is / to its type

end of course #1