Programming in Python[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Academic year 2025/26, I semester

Lecture V: Algorithms with loops

GCD

GCD

Two unequal numbers being set out, and the less being continually subtracted in turn from the greater, if the number which is left never measures the one before it until an unit is left, the original numbers will be prime to one another. [...] But, if CD does not measure AB, then, the less of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it. [...]

[Euclid *"Elements"*, Book VII, Prop. I, II (c. 300 BC)]

GCD

Two unequal numbers being set out, and the less being continually subtracted in turn from the greater, if the number which is left never measures the one before it until an unit is left, the original numbers will be prime to one another. [...] But, if CD does not measure AB, then, the less of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it. [...]

[Euclid *"Elements"*, Book VII, Prop. I, II (c. 300 BC)]

```
a: int = 420
b: int = 240

# invariant
# GCD(a, b) == GCD(|a - b|, a)
# GCD(a, b) == GCD(|a - b|, b)
while a != b:
  if a > b:
    a = a - b
  else:
    b = b - a

print(a)
```

Loops can be difficult to understand

When you have loops, understanding the code can be a difficult task and the only general strategy is to track the execution.

```
# This is known as Collatz's procedure
n = ...
while n > 1:
  if n % 2 == 0:
    # if the remainder of division by 2 is 0, i.e. n is
    ↪  even
    n = n / 2
  else:
    n = 3*n + 1
```

We know (by empirical evidence) that it ends for all $n < 2^{68} \approx 10^{20}$, nobody is able to predict the number of iterations given any `n`.
With loops it is also hard to exploit parallel execution.

Learn to write loops can be hard

When you write a loop, you should have in mind two related goals:

1. the loop must terminate: this is normally easy with `for` loops (when the finite collection ends, the loop ends also), but it can be tricky with `while`s (remember to change something in the condition);
2. the loop repeats something: the programmer should be able to write the "repeating thing" in a way that makes it equal in its form (but probably different in what it does).

The second part (technically known as loop invariant) is the hardest to learn, since it requires experience, creativity, and ingenuity.

Summary

In Python3

- Variables are names to refer to objects;
- Objects are elements of types, which define the operations that make sense on them;
- Therefore, the basic instructions are the assignment (bind a name to an object), the proper operations for each object, and the commands to ask the services of the operating system;
- One can alter the otherwise strictly sequential execution of instruction with control flow statements: `if`, `for`, `while`.

Remember that in python3, indentation matters (it is part of the syntax).

Proper operations

- On objects one can apply binary and unary operators: `2 * 3 -(-5.0) not True 'foo' + 'bar'`...
- There also built-in functions like `max(8,5,6)`, the full list is here: `https://docs.python.org/3/library/functions.html`
- (syntactically, commands like `print` or `input` cannot be distinguished from other built-in functions)
- Every object has `methods` that can be applied with the so called dot notation: `(3.2).is_integer() 'foo'.upper() 'xxx'.startswith('z')`; the list of which methods an object has is given by `dir(object)`.

Definition of functions

As variables are names for objects, one can also name fragments of code:

```
def cube(x: int) -> int:
    square = x * x
    return square * x
```

Now we have a new operation `cube`, acting on `int`s: `cube(3)`. Type hints are optional (and ignored, you can call `cube(3.2)` or `cube('foo')`), but very useful for humans (and tools like `mypy`).

```
# Equivalent
def cube(x):
    square = x * x
    return square * x
```

Naming helps solving

The tower of Hanoi
https://www.mathsisfun.com/games/towerofhanoi.html

Describe the moves for a solution

Recursive thinking is a powerful problem solving technique and it can be translated to Python thanks to recursive calls.
Hanoi moves $A \to C$:

- In A there is just one disk: move it to C
- Otherwise in A there are n disks (> 1):
  - leap of faith! I suppose to know the moves needed to move $n - 1$ disk; then
    - apply this (supposed) solution to move $n - 1$ disks from A to B (leveraging on C, empty, as the third pole)
    - move the last disk from A to C
    - apply the (supposed) solution to move $n - 1$ disks from B to C (leveraging on A, now empty, as the third pole)

This implicit description solve the problem! Finding a non-recursive solution is possible but not that easy.

In Python


```python
def hanoi(n: int, from_pole: str, to_pole: str,
↪  tmp_pole: str):
  if n == 1:
    print('Move 1 disk from ' + from_pole + ' to ' +
    ↪  to_pole)
  else:
    hanoi(n - 1, from_pole, tmp_pole, to_pole)
    hanoi(1, from_pole, to_pole, tmp_pole)
    hanoi(n - 1, tmp_pole, to_pole, from_pole)

hanoi(3, 'A', 'C', 'B')
```