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

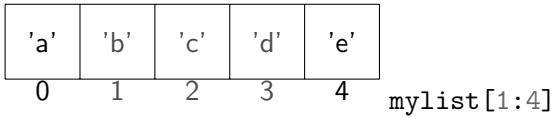
Lecture IX: Working with abstractions

Status of the homework

	accepted	done
One triangle	19	10
Triangle kinds	14	5
Newton square root	10	4
Pythagorean triplets	15	7
Sonar	13	3

List slices

- List slicing is a powerful feature in Python that allows you to create a new list by extracting a portion of an existing list.
- It's like cutting out a piece of a list you specify the starting and ending indices (exclusive of the end index).
- Syntax: `mylist[start:end]`
- `start` (optional): The index where the slice begins (inclusive). Defaults to 0.
- `end` (optional): The index where the slice ends (exclusive). Defaults to the end of the list.



Examples


```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Slice from index 2 to 5 (exclusive):
slice1 = my_list[2:5]  # [2, 3, 4]

# Slice from the beginning to index 3 (exclusive):
slice2 = my_list[:3]   # [0, 1, 2]

# Slice from index 6 to the end:
slice3 = my_list[6:]   # [6, 7, 8, 9]

# Create a copy of the entire list:
slice4 = my_list[:]  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```


More slicing

`my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

- Step Size: You can add a third argument to specify a step size (e.g., `mylist[start:end:step]`). `my_list[1:8:2]` creates a slice starting at index 1, going up to (but not including) index 8, with a step size of 2. `[1, 3, 5, 7]`
- Negative Indices: Indices can be negative, counting from the end of the list. -1 is the last element, -2 is the second to last, etc. `my_list[-3:]` takes the last three elements of the list. `[7, 8, 9]`; `my_list[:-1]` gets all elements except the last.

A slice is always a new object


```
x = [100, 200, 300, 400, 500]

slice = x[1:4]

slice[0] = 666

print(f'{x=} {slice=}')

# x=[100, 200, 300, 400, 500] slice=[666, 300, 400]
```


Procedural abstraction

Procedural abstraction is key for our thinking process (remember the power of recursion, for example): giving a name to a procedure/function enhances our problem solving skills.

```
def sum_range(a: int, b: int) -> int:
    """Sum integers from a through b.

    >>> sum_range(1, 4)
    10

    >>> sum_range(3, 3)
    3
    """
    assert b >= a
    result = 0
    for i in range(a, b+1):
        result = result + i
    return result
```

Another "sum"

This is very similar...

```
def sum_range_cubes(a: int, b: int) -> int:
    """Sum the cubes of the integers from a through b.

    >>> sum_range_cubes(1, 3)
    36

    >>> sum_range_cubes(-2, 2)
    0

    """
    assert b >= a
    result = 0
    for i in range(a, b+1):
        result = result + cube(i) # cube(i: int) -> int
        ↪  defined elsewhere
    return result
```


Another "sum"

This is also very similar...

$$\frac{1}{a\cdot(a+2)} + \frac{1}{(a+4)\cdot(a+6)} + \frac{1}{(a+8)\cdot(a+10)} + \cdots + \frac{1}{(b-2)\cdot(b)}$$

(Leibniz: $\frac{1}{1\cdot3} + \frac{1}{5\cdot7} + \frac{1}{9\cdot11} + \cdots = \frac{\pi}{8}$)

```
def pi_sum(a: int, b: int) -> float:
    """Sum 1/(a(a+2)) terms until (a+2) > b.

    >>> from math import pi
    >>> abs(8*pi_sum(1, 1001) - pi) < 10e-3
    True

    """
    assert b >= a
    result = 0.0
    for i in range(a, b+1, 4):
        result = result + (1 / (i * (i + 2)))
    return result
```


Can we abstract the similarity?


```
from collections.abc import Callable

Num = int | float

def gen_sum(a: int, b: int, fun: Callable[[int], Num], step: int = 1) -> Num:
    """Sum terms from a through b, incrementing by step.

    >>> gen_sum(1, 4, lambda x: x)
    10

    >>> gen_sum(1, 3, lambda x: x**3)
    36

    >>> from math import pi
    >>> abs(8*gen_sum(1, 1000, lambda x: 1 / (x * (x + 2)), 4) - pi) < 10e-3
    True

    """

    assert b >= a
    result = 0.0
    for i in range(a, b+1, step):
        result = result + fun(i)
    if isinstance(result, float) and result.is_integer():
        return int(result)
    return result
```


The huge value of procedural abstraction

It is worth to emphasize again the huge value brought by **procedural abstraction**. In Python it is not mandatory to use procedures/functions: the language is designed to be used also for *on the fly* calculations.

```
x = 45
s = 0
for i in range(0, x):
  s = s + i
```

This is ok, but it is not encapsulated (in fact, since encapsulation is so important you can at least consider it encapsulated in file which contains it)

- the piece of functionality is not easily to distinguish

  it could be intertwined with other unrelated code

```
x = 45
a = 67 # another concern
s = 0
for i in range(0, x):
  s = s + i
print(a) # another concern
```

- the goal is not explicit, which data are needed, what computes
- it's hard to reuse even in slightly different contexts

Encapsulate the functionality

```
def sum_to(x: int) -> int:
    assert x >= 0
    r = 0
    for i in range(0, x):
      r = r + i
    return r

s = sum_to(45)
```

- It gives to our mind a "piece of functionality", the interpreter we are programming is now "able" to do a new thing that can be used **without thinking about the internal details**
- It makes clear which data it needs (an integer, ≥ 0 if we add also an assertion or a docstring)
- It makes clear that the interesting result is another integer produced by the calculation
- It can be reused easily and safely

Homework

- https://classroom.github.com/a/Auwejr2m