Programming and Algorithms 1

Degree in Bioinformatics, UPF

José Luis Balcázar, Lluís Padró

Dept. Computer Science, UPC

Fall 2021 and Winter 2022

Contents

Index

Evaluation and Support

Course spreads on during two terms instead of finishing with a final exam in December.

3rd exam	February 8th.
4th exam	March 21st (plans as of today).

- Slides available at:
`http://www.cs.upc.edu/~balqui/slides_ProgAlg1_2022_winter.pdf`
(live link, also available at `http://aula.esci.upf.edu`).
- The slide deck is again "live" and evolving; I will issue warnings in class whenever an update is uploaded.
- Additional zip file with snippets also in the Aula Virtual.
- "Book": `http://composingprograms.com/about.html`

Algorithms and Programs

Algorithm:	Instructions so precise that can be carried out without understanding. Precision necessary up to an inhuman extent.
Program:	Algorithm expressed in a precise, conventional code, called a "programming language".
Expression:	Composition of functions (a known concept), it is the construction that allows for nontrivial computations to take place: the result of a function is used as argument for another function.
Variable:	Association of a name to some value.
Type:	Way of identifying the repertory of functions available on a given value.

Functions Old and New

- Many pre-existing functions available.
- We may create as many new ones as convenient.
- We must pay attention to hitting right the arities:
  - each function must receive the number of arguments it expects, and
  - each argument must be of an appropriate type.
- Some common (often binary) functions have an associated operator: a symbol written between the arguments (infix notation). This may require parentheses to disambiguate. Particularly relevant: comparison operators.
- Object-oriented notation allows us to use the same function name with different meanings.

Types

Unstructured:	integers, floats, booleans;
Structured:	strings, tuples, ranges, files, lists, sets, dictionaries.

- Changing types, or explicit casting:
  - To convert e.g. between `str` and numeric values.
  - Some string operations also useful for such tasks — but nontrivial to select which are the ones to be applied.
- Coercion: implicit, silent casting.
  - Numeric values get coerced into larger numeric spaces when necessary.
  - Most anything gets coerced into `bool` when necessary.

Modular Thinking

Decompose your problems into simpler problems.

> Very carefully, because sometimes the problem “looks” simpler but is not.

Organize them:

- Define new functions.
- Import useful functions from external modules.
- Consider joining together related functions as methods of a class.

Object-Oriented Approach

Object-oriented notation allows us to use the same function name with different meanings.

- One of the arguments becomes the “owner” of the function:
  - thus objects own functions;
  - functions owned by an object are called its methods;
  - calls to a method are as if the owner is the first argument;
  - however, the syntax is:
    the owner precedes the name of the method, with an intervening dot.
- Objects of the same class have the same repertory of methods.

Details to Review

Mostly on your own

- Quote schemes for strings:
  - one, two, or three at each end?
  - single or double quotes?
  - quotes inside quotes, newlines inside quotes...
- Rules for identifiers.
- Arithmetic and comparison operators and their overloading.
- String methods, e.g. `join()`.
- Function's and module's docstrings.
- How to `doctest` a module, and how to set up around doctests your own programming process.
- Indexing tuples: strides, slicing...
- Conceptual differences between tuples, ranges, and strings.

Algorithmic Schemes

- Syntactical aspects:
  - Alternatives: `if`, `elif`, `else`;
  - `for` loops;
  - `while` loops.
- Sequence processing:
  - Iterables (including all structured types),
  - transforming iterables by comprehension,
  - the traversal or full scan algorithm scheme,
  - the linear search algorithm scheme;
  - usage of files.

Files

Communicating with the rest of the computer: files as documents in the hard drive and the special files `stdin` and `stdout`.

- Operations on `stdout`: usually delegated to `print()`, but generic file method `write` available for occassional usage.
- Operations on `stdin` or on other files:
  - `readline()`;
  - files are iterable so we can run a `for` loop on them;
  - for reading in a single, occassional string we can use `input()`.
- Remember:
  - to `open()` all files different from `stdin` or `stdout` and either
  - to `close()` them explicitly or
  - to use them within a context manager instruction `with`, which will take care of the closing on itself.

Mutability

Possibility of freely modifying individual components of a data structure while keeping all the rest unchanged.

Immutable sequences:

- Tuples,
- strings,
- ranges,
- files.

Mutable sequences:

- Lists.

List Usage Examples, I

Design a program that works as follows:

> For a given word *w* and a file containing a text, find all the words that appear in the text exactly after *w*; provide them in the order they appear.

We try it out on the `dq5.txt` file with words obtained with `input()`.

If you are faster than your colleagues and finish your task early, tackle variants; for instance, words that appear exactly before *w*, or assuming that *w* is not given explicitly but instead by its position in the text, or...

(See `e0100_0_after.py` for a solution.)

List Usage Examples, II

Matrices can be represented as lists of rows,
where rows are, in turn, also lists.

Given such a matrix, is any entry a zero?

```
'''
>>> zero_in_matrix([[1, 2, 3], [3, 3, 3], [3, 2, 1]])
False
>>> zero_in_matrix([[1, 2, 3], [3, 0, 3], [3, 2, 1]])
True
'''
```

List Usage Examples, II

Matrices can be represented as lists of rows,
where rows are, in turn, also lists.

Given such a matrix, is any entry a zero?

```
'''
>>> zero_in_matrix([[1, 2, 3], [3, 3, 3], [3, 2, 1]])
False
>>> zero_in_matrix([[1, 2, 3], [3, 0, 3], [3, 2, 1]])
True
'''
```

See `e0102_0_z_matrix.py` which is:

```
def zero_in_matrix(m):
    for row in m:
        for elem in row:
            if elem == 0: return True
    return False
```

Index

Dictionaries

Tabulated partial functions

Partial functions correspond to mappings.

For instance, we can map a set of integers, the domain of the partial function, into strings:

```
    { 5 : "abc", 31 : "cdefgh" }
```

The particular domain for this example is the set $\{5, 31\}$.

Dictionaries

Tabulated partial functions

Partial functions correspond to mappings.

For instance, we can map a set of integers, the domain of the partial function, into strings:

```
{ 5 : "abc", 31 : "cdefgh" }
```

The particular domain for this example is the set $\{5, 31\}$.

The elements of the domain of a dictionary are called keys.

- One way to write dictionaries is by explicit pairs, indicating, for each key, its associated value.
- Key and value are separated by a colon ':' and pairs are separated by commas.
- Enclose the whole thing in curly braces as above.
- This simple usage is, however, not very common.

Possibilities for Keys and Values

The role of mutability

We can use as keys

- integers,
- floats,
- but also strings,
- or all sorts of tuples,
- or frozensets (to be studied later)...

and more generally any immutable type; and the associated values can be anything, also mutable types.

The empty dict, that is, the empty partial function, is the one on empty domain. It can be written { } and can be also obtained as `dict()`.

Dictionary Keys and Values

Usage syntax

Given a `dict d`, we can find the value associated to any key `k`:

```
d[k]
```

Dictionaries are mutable! We can assign a new value to a key while leaving the rest of the dict unchanged:

```
d[k] = v
```

We can iterate on them; since Python 3.7, `dict`'s are iterated on in guaranteed key insertion order.

The function `sorted` is also available; default configuration is to sort on the keys.

Dictionary Usage Examples

A common way of using dictionaries; see `e0105_*_lineoccurs.py`

Most often, we construct dictionaries one key at a time.

Example

In which line did each word appear last?

Dictionary Usage Examples

A common way of using dictionaries; see e0105_*_lineoccurs.py

Most often, we construct dictionaries one key at a time.

Example

In which line did each word appear last?

```
last_lines = dict()
line_cnt = 0
with open("dq5.txt") as f:
    for line in f:
        "update line number for words in this line"
        for word in line.split():
            last_lines[word] = line_cnt
        line_cnt += 1
print(last_lines)
```

(Then, take advantage of the fact that a dict is iterable to go through the keys and print out better output.)

Modifying Values and Initialization

See `e0110_*_countoccurs.py`

Often, we want to update a value in a dict, for instance:

```
d[k] = d[k] * d[k]
```

This needs an “if” to cover for the first arrival of k:

```
if k in d:
    d[k] = d[k] * d[k]
else: ...
```

or

```
d[k] = d[k] * d[k] if k in d else ...
```

For instance: let's count occurrences of each word.

Default Values

Some handy variants

Useful alternatives:

- default dictionaries in the `collections` module: upon creation, one explains once and for all what the default is upon accessing an undefined key

  (effectively turning the partial function into a total function);
- default dictionaries of integers have a separate implementation: the class `Counter`;
- other options to explore on yourself.

Further Dictionary Exercises, I

Similar to the exercise on lists in a previous day.
Design a program that works as follows:

> Given a file containing a text, find, for each word *w* in the text, all the words that appear in the text exactly after *w*; provide them in the order they appear.

We try it out on the `dq5.txt` file.

Solve it first with a `dict()`; then, if you wish, explore how a solution importing `defaultdict` would go. Again, further variations are possible.

Further Dictionary Exercises, II

With the same protein of a few weeks ago

Fish back into your practice folder the `1bl8.pdb` protein to be found among the Programs and Data Files zip file.

(Source: `http://files.rcsb.org/view/1bl8.pdb`)

Recall script `e0070_0_count_atoms.py` that counts atoms.

1. Create a new script that gives the counts broken down by atom.
2. Create a new script that gives the atoms that appear before and after each atom.
3. Simplify these scripts using default dictionaries from the `collections` module; for the first one, try alternatively the Counter class in the same module.

Programming Project

A brief experience about (non-professional) game design

Video Games are these days a powerful industry.

- No free / open-source platform is able to provide industry-grade speed.
    - Each company has its own propietary "game engine" that handles a variety of primitive graphics operations,
    - while extracting the maximum potential from the hardware.
    - These game engines offer a platform on top of which the actual games are programmed, often in Python.
- A good open-source platform for starters: `pygame`.
- We learn a bit of it today:
    - As usual, we just scratch the surface.
    - Warning: we will be "refreshing the whole screen" several times per second, which is very inefficient;
    - how to accelerate that (using "minimal rectangle blits") should be the first thing to learn if you want to go beyond.
- Demo of an extremely simple game, organized in seven steps.

Rules

For the project

Goal: convincing you all of spending more time programming.

1. Organize yourselves into teams of two or three people.
2. Each team designs and programs their own video game.
3. Before embarking on the programming task, each team discusses with me the ideas. I might try to dissuade you of some goals if they look too difficult to me.
4. All sources of inspiration are valid. Previous years have seen many variants of dogfighting aircraft or spaceships, classical space invaders, of course Pong, driving snakes or other approaches to fruit harvesting while avoiding some mishaps. . .
5. To be delivered in early March.
6. Influence on the grade only upwards; hence, optional.
7. However, everyone is encouraged to try and participate as the only way of learning to write programs is writing programs.

Sets

The familiar notion

Sets are what we know of from typical set-theoretic math.

Main property:

An element either belongs to the set, or does not belong to it.

- No discussion of "how many times" it is there.
- No discussion about in which order the elements are.

Hence: Handy for removing duplicates!

Warning: Only immutable objects can be members of sets.

Fully standard set operations.

Only detail to care for:

since sets are mutable,

- does each operation change the set?
- or does it create a new set instead?

Operations Available on Sets

Creating sets:

- `s = set()` initializes s as the empty set;
- `s = set(c)` initializes s as a set with the elements of `c`, where `c` is any iterable (like a tuple or a list);
- `s.add(e)` adds one element to the set.

Then, two versions of each operation:

- `s1.union(s2)` returns a new union set
  (`s1` and `s2` remain as they were);
- `s1.update(s2)` leads to `s1` becoming the union
  (`s2` remains as it was, nothing is returned);
- likewise with other operations like intersection and difference.
- Comparisons are interpreted as set inclusions.

Of course, sets are iterable; and any iterable is accepted as second argument of union, update, and similar operations.

Set Usage Examples

See e0120_0_primes3and5.py

We construct a set of at least n numbers that can be obtained through products involving only primes 3 and 5.

Initially we write the list [1] with just the int 1 inside, and transform it into a set:

```
s = set([1])
while len(s) < n:
  s.update([ 3*i for i in s ] + [ 5*i for i in s ])
for i in s:
  print(i)
```

Compare with what happens if s is instead a list.

Detecting duplicates in a file: compare traversal with linear search, compare list to set (availability of snippets: next week).

Frozensets

Immutable versions, useful to index a `dict`; see `e0130_*_countsets.py`

Sets are mutable.

Sometimes one needs an immutable variant: `frozenset`.
The most common reasons are the need to use them

- as `dict` keys or
- as `set` members.

Example: We are given sets of words to read, each set in a line.
How many times did each set appear?

```
set_count = dict()
for line in stdin:
  s = set( line.split() )
  if s in set_count:        # fails!
    set_count[s] += 1
  else:
    set_count[s] = 1
```

Frozensets

Immutable versions, useful to index a `dict`; see `e0130_*_countsets.py`

Sets are mutable.

Sometimes one needs an immutable variant: `frozenset`.
The most common reasons are the need to use them

- as `dict` keys or
- as `set` members.

Example: We are given sets of words to read, each set in a line.
How many times did each set appear?

```
set_count = dict()
for line in stdin:
  s = frozenset( line.split() )
  if s in set_count:
    set_count[s] += 1
  else:
    set_count[s] = 1
```

Integer Keys

Dictionary or list?

When we associate info to integers, we have two options:

- a `dict` with `int` keys,
- a `list`, using the integers as positions.

Integer Keys

Dictionary or list?

When we associate info to integers, we have two options:

- a `dict` with `int` keys,
- a `list`, using the integers as positions.

You must master both!

Integer Keys
Dictionary or list?

When we associate info to integers, we have two options:

- a `dict` with `int` keys,
- a `list`, using the integers as positions.

You must master both!

- Retrieving the values is fast in both cases, the list being faster by a tiny margin.
- The list requires to decide on an upper bound for the keys.
- Hence, a list may use much more space, far more than necessary, and this may slow it down.
- Ask yourself whether the range of usage is worth it.

Challenge: set up your own timing experiments about this question.

Index

Iterators, I

Behind the "iterable" data structures

Iterators are objects that implement one particular method that "provides the next element" (if it exists):

> keep calling `next(...)` to traverse the object.

We know of several examples of iterable types of objects: they have an associated iterator

> (usually obtained as `iter(...)`).

- tuples, including strings,
- files,
- lists,
- dictionaries,
- sets and frozensets,
- range objects. . .

Here range objects are slightly different: "not materialized", not explicit in memory but just a program generating them.

How does `next(...)` work (and many others)

A convention

Many types of objects have methods or attributes that normal programming is not expected to employ, but are there to help other constructions.

Their names start and end with double underscores.

An example: the `__next__()` method of iterators.

(Other examples: `__len__()`, `__str__()`, `__name__()`, `__main__`,. . .)

The external function (like `next(...)`) simply calls the method.

Iterators, II

Useful iterators in the `itertools` module

Ready-made iterators to

- repeat indefinitely some element (or a fixed number of times),
- count indefinitely from some int onwards,
- cycle indefinitely over the elements of another iterator,
- traverse the cartesian product of two iterables,
- produce permutations or combinations. . .

Some iterators may iterate indefinitely!

- Be careful.
- The `break` instruction inside a loop may be handy.
- Also the built-in function `zip`.

A useful transformation: `reversed()`

- iterator for backwards traversal of a sequence.

Iterators, III

Sources

- Iterables

  (that provide their corresponding iterator);
- generators, that is, non-materialized, programmatic iterables:
  - ranges;
  - the `enumerate` generator;
  - comprehensions;
  - explicitly programmed iterators;
  - ...

Generators, I

Or: Iterators IV

We have seen already comprehensions: ways of transforming iterators, like

```
str(i) for i in range(10)
int(s) * int(s) for s in line.split()
```

Normally we either run a `for` loop on them, or “materialize” them into a list or tuple, but we could as well grab the iterator and keep calling `next(...)`.

Generators, II

Compact forms of programming iterators explicitly

We can program our own generators using the instruction `yield`:

```
def stdin_words():
  from sys import stdin
  for line in stdin:
    for w in line.split():
      yield w
```

Generators, III

Example: the Fibonacci sequence, in generator form

```
def fibonacci():
  a, b = 0, 1
  yield a
  yield b
  while True:
    a, b = b, a + b
    yield b
```

Careful: it would loop forever, so, as we said before, the program that calls it must control how long it runs.

Again you can control it with `for/if/break` or with the built-in function `zip`, for example.

`Further explanations about iterators and generators.`

Index

Index