TDLOG – session 2
Modeling

Xavier Clerc – `xavier.clerc@enpc.fr`
Clémentine Fourrier – `clementine.fourrier@inria.fr`
Michael Herbst – `michael.herbst@inria.fr`

30 September 2019

Upload your work to educnet on 5 October 2019 at the latest

This document is made of three parts : a part about exercises, and two annexes about the interpreter and some predefined functions useful for the exercises.

1 Modeling a game

The objective of this session is to define the classes and functions needed to represent a simple game. You are expected to produce a code that is both readable, and handles the various error cases.

1.1 Informal specification

The object of the game is to move a character on a grid similar to the one of Figure 1.

Each cell of the grid may contain one of the following elements :
— the character (*e.g.* cell $\langle 1, 1\rangle$) ;
— a wall (*e.g.* cells $\langle 4, 0\rangle$ and $\langle 7, 7\rangle$) ;
— a hole (*e.g.* cells $\langle 4, 6\rangle$ and $\langle 11, 1\rangle$) ;
— a crate (*e.g.* cells $\langle 2, 6\rangle$ and $\langle 9, 3\rangle$) ;
— a door (*e.g.* cell $\langle 14, 6\rangle$) ;
or be empty.

The goal of the character is to reach the door ; when it reaches the door, the game is over. The character can move either horizontally or vertically, one cell at a time if the destination cell is empty. It can also push a create as shown on Figure 2 (the cell next to the crate must be empty), following the same directions. Finally, it can fill a hole by pushing a crate into it as shown on Figure 3 ; in this case, both the crate and the hole disappear.

FIGURE 1 – Example of a grid.

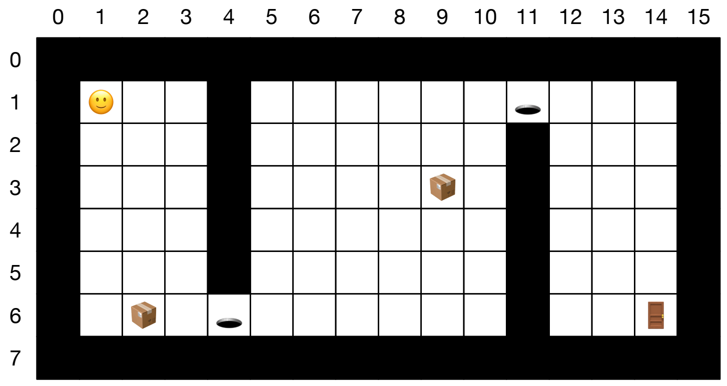


FIGURE 2 – Pushing a crate.

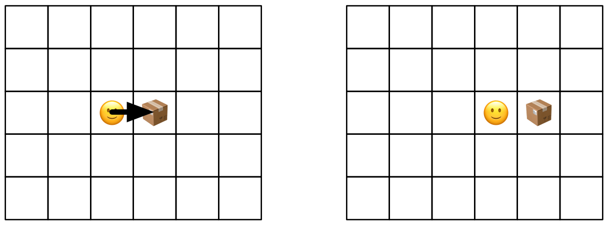
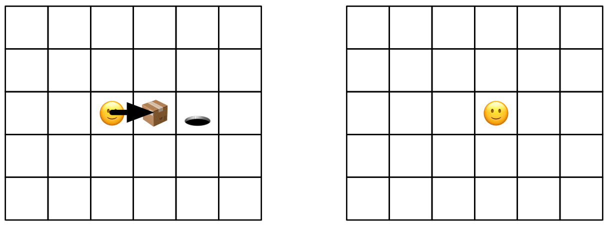


FIGURE 3 – Filling a hole.



1.2 Implementation

The first step is to define the elements needed in order to be able to represent a grid with its contents. A grid must be at least 5-cell wide and 5-cell high, must contain one character and one exit, and have walls on the borders as shown on Figure 1.

1.3 Printing

The second step is to define the elements needed to print the grid as bare text, using the following encoding :

— `1` for the character ;
— `#` for a wall ;
— `o` for a hole ;
— `*` for a crate ;
— `@` for the door ;
— ` ` (space) for an empty cell.

The grid from Figure 1 is hence printed as follows :

```
################
#1  #      o   #
#   #      #   #
#   #    * #   #
#   #      #   #
#   #      #   #
# * o      #  @#
################
```

1.4 Interaction

The third step is to define the elements needed to interpret the orders from a player as typed on the keyboard. An order is a string encoding the directions of the successive moves of the character. The directions are specified as follows :

— `>` for an horizontal move towards right ;
— `<` for an horizontal move towards left ;
— `^` for a vertical move towards top ;
— `v` for a vertical move towards bottom.

This string `>>v` hence means that the player wants to move the character twice towards the right and one towards the bottom. If a move is not possible, the direction is simply ignored. The following string is a solution to the grid of Figure 1 :

```
vvvvv>>>>>>>>^^^^<^>>>>>>vvvvv
```

The complete program is expected to repeat the following steps until the game is over :
- print the grid ;
- let the player enter a string ;
- interpret the string as specified above.

2 Annex : Python interpreter

In this section, we present various statements you can execute inside the interpreter in order to get a feeling of how *Python* works, if you are not familiar with it yet.

Once the interpreter is started, it displays a short message about the version, and then a prompt waiting for you to type code fragments to be evaluated. The default prompt is `>>>`. When you type a line that (possibly) expects additional lines, the interpreter changes the prompt to `...` in order to indicate that it expects something. Obvious, you have to use the correct indentation, and typing only "enter" to a `...` prompt indicates that your code fragment is complete and ready to be evaluated.

Basically, the interpreter can be used as a calculator, with the possibility to store results into variables :

```
>>> 3 + 4
7
>>> a = 3
>>> b = a + 5
>>> print(a, b)
3 8
```

It is also possible to define functions and then use them in expressions :

```
>>> def square(x):
...   return x * x
...
>>> print(square(5))
25
```

In order to iterate over the elements of a list or string, a `for` loop is used (beware of indentation) :

```
>>> for e in [1, 2, 3]:
...   print(e)
...
1
2
3
```

```
>>> for ch in "string":
...   print(ch)
...
s
t
r
i
n
g
```

When the *iterable* is a dictionary, the iteration is over its keys :

```
>>> for clef in { 0:"zero", 1: "one", 2:"two" }:
...   print(clef)
...
0
1
2
```

If you want to repeat a code n times, the simplest way is to use the `range(n)` function :

```
>>> for i in range(5):
...   print("****", i)
...
**** 0
**** 1
**** 2
**** 3
**** 4
```

Strings can be concatenated through the `+` operator, and the concatenation of a string list using a separator is made through the `join` function :

```
>>> print("first string" + " " + "second string")
first string second string
>>> print(", ".join(["one", "two", "three"]))
one, two, three
```

When working with sets or dictionaries, the `in` and `not in` operators allow to test the membership of a value to the set or dictionary keys :

```
>>> e = {2, 4, 5}
>>> print((2 in e), (5 not in e))
True False
```

Comprehensions are short, yet readable, ways to define lists, sets, or dictionaries :

```
>>> [ i * i for i in range(10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> def mult7(x): return (x % 7) == 0
...
>>> { i for i in range(100) if mult7(i) }
set([0, 98, 35, 70, 7, 42, 77, 14, 49, 84, 21, 56, 91, 28, 63])
>>> { i: i*i for i in range(10) }
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

3 Annex : commonly-used functions and methods

This section lists the functions and methods that one should know in order to be able to work on the exercises or write simple programs. However, the descriptions are purposely short ; a comprehensive documentation (about all functions, classes and methods of the standard library) is available from either the online reference documentation, or through the `help(...)` function of the interpreter.

3.1 Main *built-in* functions

Some classes, whose constructors can be seen as conversion functions, converting the only received parameter :
- `bool` ;
- `int` ;
- `float` ;
- `complex` ;
- `list` ;
- `dict` ;
- `set` ;
- `frozenset` ;
- `str` (for *string*s).

Some functions for list handling :
- `len(`l`)` returns the length of l ;
- `map(`f`, `l`)` returns an iterator equivalent to `[` $f(e_0), f(e_1), \ldots, f(e_n)$ `]` where l = `[` $e_0, e_1, \ldots, e_n$ `]` ;
- `min(`l`)` returns the smallest element from l ;
- `max(`l`)` returns the largest element from l ;
- `sum(`l`)` returns the sum of the elements from l.

The `range` function accepts one to three parameters, and returns a generator of integers :
- a first optional parameter specifying the lower bound (inclusive), defaulting to 0 ;
- a second parameter specifying the upper bound (exclusive) ;

— a third optional parameter specifying the step between elements.

The `sorted` function accepts one to three parameters, and returns a sorted list (using ascending order) :

— the first parameter is the *iterable* (*e.g.* a list) to sort ;

— an optional parameter, `key`, allowing to specify the sorting key ;

— an optional parameter, `reverse`, allowing to reverse the order (thus using descending order).

The `print` function accepts any number of parameters and prints them using single spaces as separators. An end-of-line character is also printed after the elements. The function accepts the named parameters `sep` (separator to be used instead of space) and `end` (replacing the end-of-line character).

The `input` function reads a string from the keyboard. It accepts an optional parameter used as a *prompt*.

The `ord` and `chr` functions respectively convert a character into an integer and an integer into a character, the integer being the code of the character.

3.2 Main methods about lists

In the following, l is a list :

— l`.append(`x`)` appends x to the list ;

— l`.insert(`i, x`)` inserts x at the position i ;

— l`.index(`x`)` returns the index of the first occurrence of x in l ;

— l`.count(`x`)` returns the number of occurrences of x in l ;

— l`.reverse()` modifies l by reversing the order of the elements ;

— l`.sort()` modifies l by sorting it, the function accepts two optional parameters, `key` and `reverse` that respectively specify the sorting key and whether the order should be reversed.

3.3 Main methods about strings

In the following, s is a string :

— s`.upper()` returns a copy of s where all letters are uppercase ;

— s`.lower()` returns a copy of s where all letters are lowercase ;

— s`.replace(`o, n`)` returns a copy of s where all occurrences of o are replaced with n ;

— s`.split(`*sep*`)` returns a list of elements of s as separated by *sep* ;

— s`.join(`*seq*`)` returns the concatenation of the elements from the sequence *seq*, s being used as the separator ;

— s`.format(...)` accepts any number of (possibly named) parameters and applies a `printf`-like formating. For example, `"x = {}, y = {}, z = {}".format(3, 5, 7)` returns the string `"x = 3, y = 5, z = 7"`. Moreover, if a parameter is named n, it is possible to reference it through `"{n}"`. A comprehensive documentation of all formating options is available at `https://docs.python.org/3/library/string.html#formatspec`.

3.4 Main methods about dictionaries

In the following, d is a dictionary :

— d`.items()` returns the set of $\langle$ key, value $\rangle$ couple of d ;

— d`.keys()` returns the set of the keys of d ;

— d`.values()` returns the list of the values of d.