Programming Fundamentals

Data Types: Working with Lists

João Correia Lopes

INESC TEC, FEUP

13 November 2018

Goals

By the end of this class, the student should be able to:

- Use the main methods available to work with lists
- Use generalised `for` loops with lists
- Describe pure functions and modifiers (that make side-effects)
- Describe type conversions (`list` and `range`)
- Use nested lists to work with matrices

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Section 5.3) [PDF]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Chapter 10) [HTML]
- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3 (RLE), 2012 (Chapter 11) [HTML]

Tips

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

Contents

A COMPOUND DATA TYPE

- So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists, pairs or tuples
- Strings, **lists**, and tuples are qualitatively different from the others because they are made up of smaller pieces
- Lists (and tuples) group any number of items, of different types, into a single compound value
- Types that comprise smaller pieces are called **collection** or **compound data types**
- Depending on what we are doing, we may want to treat a compound data type as a single thing

LISTS AND FOR LOOPS

- The `for` loop also works with lists, as we've already seen
- The generalized syntax of a `for` loop is:

```
for <VARIABLE> in <LIST>:
    <BODY>
```

- "For (every) friend in (the list of) friends, print (the name of the) friend"

```
friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
for friend in friends:
    print(friend)
```

LIST EXPRESSIONS IN FOR LOOPS

- Any list expression can be used in a `for` loop
- `enumerate` generates pairs of both *(index, value)* during the list traversal

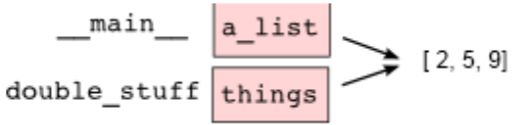
```
1  xs = [1, 2, 3, 4, 5]
2
3  for (i, val) in enumerate(xs):
4      xs[i] = val**2
```

⇒ https://github.com/fpro-admin/lectures/blob/master/13/for-lists.py

LIST PARAMETERS

- Passing a list as an argument actually passes a *reference* to the list, **not a copy** or clone of the list
- So parameter passing creates an alias[1]

```
1  def double_stuff(things):
2      """ Overwrite each element in a_list with double its value. """
3      ...
4  a_list = [2, 5, 9]
5  double_stuff(a_list)
```



[1] The caller has one variable referencing the list, and the called function has an alias, but there is only one underlying list object

LIST METHODS

- The dot operator can also be used to access built-in methods of list objects

```
>>> mylist = []
>>> mylist.append(5)       # Add 5 onto the end of mylist
>>> mylist.append(12)
>>> mylist
[5, 12]

>>> mylist.insert(1, 12)  # Insert 12 at pos 1, shift others
>>> mylist.count(12)      # How many times is 12 in mylist?

>>> mylist.extend([5, 9, 5, 11])
>>> mylist.index(9)       # Find index of first 9 in mylist
>>> mylist.reverse()
>>> mylist.sort()
>>> mylist.remove(12)     # Remove the first 12 in mylist
```

PURE FUNCTIONS AND MODIFIERS

- As seen before, there is a difference between a pure function and one with side-effects
- Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**
- A **pure function** does not produce side effects
  - It communicates with the calling program only through parameters, which it does not modify, and a return value

⇒ https://github.com/fpro-admin/lectures/blob/master/13/double_stuff.py

FUNCTIONS THAT PRODUCE LISTS

- Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
   create a new element
   append it to result
return the result
```

```
def primes_lessthan(n):
    """ Return a list of all prime numbers less than n. """
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
return result
```

FUNCTIONS THAT PRODUCE LISTS

- Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
   create a new element
   append it to result
return the result
```

```
def primes_lessthan(n):
    """ Return a list of all prime numbers less than n. """
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
return result
```

STRINGS AND LISTS

- Two of the most useful methods on strings involve conversion to and from lists of substrings
- The `split` method breaks a string into a list of words
- By default, any number of whitespace characters is considered a word boundary
- An optional argument called a **delimiter** can be used to specify which string to use as the boundary marker between substrings
- The inverse of the `split` method is `join`
- You choose a desired **separator** string and join the list with the glue between each of the elements

⇒ https://github.com/fpro-admin/lectures/blob/master/13/strings.py

LIST

- Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list

```
>>> letters = list("Crunchy Frog")
>>> letters
["C", "r", "u", "n", "c", "h", "y", " ", "F", "r", "o", "g"]
>>> "".join(letters)
'Crunchy Frog'
```

RANGE

- One particular feature of `range` is that it doesn't instantly compute all its values:
  - it "puts off" the computation, and does it on demand, or "lazily"
  - We'll say that it gives a **promise** to produce the values when they are needed
  - This is very convenient if your computation short-circuits a search and returns early

⇒ https://github.com/fpro-admin/lectures/blob/master/13/lazy-eval.py

LIST AND RANGE

- You'll sometimes find the lazy range wrapped in a call to list
- This forces Python to turn the lazy promise into an actual list

```
>>> range(10)           # Create a lazy promise
range(0, 10)

>>> list(range(10))     # Call in the promise, to produce a list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Looping and lists

- Computers are useful because they can repeat computation, accurately and fast
- So loops are going to be a central feature of almost all programs you encounter

Tip: Don't create unnecessary lists

Lists are useful if you need to keep data for later computation.
But if you don't need lists, it is probably better not to generate them.

⇒ https://github.com/fpro-admin/lectures/blob/master/13/sums.py

LOOPING AND LISTS

- Computers are useful because they can repeat computation, accurately and fast
- So loops are going to be a central feature of almost all programs you encounter

TIP: DON'T CREATE UNNECESSARY LISTS

Lists are useful if you need to keep data for later computation.
But if you don't need lists, it is probably better not to generate them.

⇒ https://github.com/fpro-admin/lectures/blob/master/13/sums.py

NESTED LISTS

- A nested list is a list that appears as an element in another list

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
>>> print(nested[3])
[10, 20]

>>>  nested[3][1]
20
```

MATRICES

- Nested lists are often used to represent matrices[2]
- For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> mx[1]        # select a row
[4, 5, 6]

>>> mx[1][2]     # extract a single element
6
```

[2] Later we will see a more radical alternative using a dictionary.

MATRICES

- Nested lists are often used to represent matrices[2]
- For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> mx[1]        # select a row
[4, 5, 6]

>>> mx[1][2]     # extract a single element
6
```

[2] Later we will see a more radical alternative using a dictionary.

EXERCISES

- Moodle activity at: LE13: Working with Lists