Python and Bioinformatics

Pierre Parutto

October 9, 2016

Contents

Chapter 1

Common Data Structures

During the fist classes, we used types representing a single piece of information: *boolean*, *integer* or *float*. We also started manipulating *string* but only as a single value and not as an ensemble of characters.

In this class we will see how to manipulate types that can represent multiple values at once. There exist multiple ways to organize multiple values together (mainly linearly and hierarchically), we call them **data structures**. We will see two main data structures present that are directly available in Python: sequences and dictionaries.

1.1 Sequences

Sequences organize values in a linear ordered manner. Each value in a sequence can be accessed by an integer representing its position in the chain. There exist two main types of sequences in Python: *string* and *list*. We will start by explaining their common features and then look at their differences.

1.1.1 Manipulating Sequences

Elements in a sequence are accessed using their positions.

Warning

In Python positions in start at **0**.

Hence the first element of a sequence is at position 0, the second at position 1 and the last at the position: "number of elements of the sequence" -1.

Definition 1 *The built-in function* `len(s: sequence) -> int` *returns the number of elements contained in the sequence* `s`.

The syntax to access an individual element in a sequence is to put the sequence name and then the position in-between square brackets. If `s` a sequence then `s[i]` accesses to the element of `s` at position `i`.

Remark

The valid positions in a sequence `s` are between `0` (first element) and `len(s)-1` (last element).

Warning

If you try to access to an invalid position `i` of a sequence `s`, for example when `i >= len(s)`, you will get an error:

```
>>> s = "ATGT"
>>> s[4]
IndexError: string index out of range
```

Remark

Accessing to a negative position **do not produce** an error, instead Python consider it as a position starting from the end of the sequence:

```
>>> s = "ATGT"
>>> s[-1]
T
>>> s[-2]
G
```

1.1.2 String

Strings are represented in Python by the type `str` and can only contain character values.

1.1.2.1 Definition

Strings are defined by an ensemble of characters in-between **double quotes**: `"..."`.

Example :

```
>>> "azadazkp"
"azadazkp"
>>> "3459034E*"
"3459034E*"
>>> type("3459034E*")
str
```

Empty String The empty string is simply two successive double quotes: `""`, it has a length of 0.

Warning

If you do not put double quotes around the string you want to define, Python will consider the string as a variable name:

```
>>> azdkj
NameError: name 'apzlpzda' is not defined
```

Warning

In Python, it is not possible to modify the elements of a string:

```
>>> s = "ATG"
>>> s[0] = "C"
TypeError: 'str' object does not support item assignment
```

1.1.3 List

A list (of type `list`) is a sequence that can contain values of any type.

1.1.3.1 Definition

Lists are defined by an ensemble of values separated by comas, in-between square brackets: `[val1, ..., valN]`.

Example :

```
>>> [1, 2, 3, 4]
[1, 2, 3, 4]
>>> ["A", "B", 1, 2.0]
["A", "B", 1, 2.0]
```

Empty List The empty list is obtained by an opening followed by a closing square bracket: `[]`, it has a length of 0.

Example :

```
>>> [1,2,3] + ["A", "B", "C"]
[1,2,3,"A","B","C"]
>>> [1,2,3] + []
```

```
[1,2,3]
```

1.1.3.2 Appending Elements

To add a new elements at the end of a list, Python provides the function `append`.

Example :

```
>>> l = [1,2,3]
>>> l.append(4)
>>> l
[1,2,3,4]
>>> l.append([])
[1,2,3,4,[]]
```

Warning

`append` does not return any value ! **Do not** do something like:
`l = l.append(elt)`. Here is an example of what happen if you do that:

```
>>> l = [1,2,3]
>>> l = l.append(4)
>>> l
None
```

Remark

`append` is not exactly a function, it uses a dotted notation: `l.append(elt)` where `l` is a list. The dot means that the function `append` will be applied to the list `l`.

1.1.3.3 Element Modification

In lists, one can directly modify the elements:

```
>>> l = [1,2,3]
>>> l[0] = False
>>> l
>>> [False, 1, 2]
```

1.1.4 Tuple

A tuple has the same properties than a list but its elements cannot be modified.

1.1.4.1 Definition

Tuples are defined by an ensemble of values separated by coma in-between parenthesis: `(val1, ..., valN)`.

Example :

```
>>> t = (1,2,"A", [4])
(1,2,"A", [4])
>>> t[0] = 2
TypeError: 'tuple' object does not support item assignment
```

Empty Tuple The empty list is obtained by an opening followed parenthesis directly followed by a closing one: `()`, it has a length of 0.

1.1.5 Concatenation Between Sequences

The concatenation between two sequences `s1` and `s2`, `s = s1 + s2` creates the new sequence `s1[0]...s1[len(s1)-1]s2[0]...s2[len(s2)-1]` of length `len(s1) + len(s2)`.

Example :

```
>>> "abc" + "def"
"abcdef"
>>> "def" + "abc"
"defabc"
>>> s1 = "123"
>>> s2 = "456"
>>> s1 + s2
"123456"
```

Remark

The empty sequence (either `""` or `[]`) is the neutral element for the concatenation (as the 0 for the addition): `s1 + [] = [] + s1 = s1`:

```
>>> [1, 2, 3] + []
[1, 2, 3]
>>> [] + [1, 2, 3]
[1, 2, 3]
```

Warning

Note that the operator + can only concatenate two strings, if any of the

two values is not of type `string`, Python will throw an error:

```
>>> 1 + "ABC"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> "ABC" + 1
TypeError: Can't convert 'int' object to str implicitly
```

Note that the errors are different: in the first case Python tells you that it does not know how to add an integer and a string. In the second case it tells you that it does not know how to convert `1` into a string.

1.1.6 Sequence Traversal

Definition 2 *Traversal: accessing all the elements of a sequence.*

The standard piece of code to perform a traversal is the following:

```
s = ... #s is some sequence
cpt = 0
while cpt < len(s):
    #s[cpt] is the element at position cpt
    cpt = cpt + 1
```

Remark

This code has to be adapted (replace the comment line by one or more lines) to the specific processing you have to perform.

1.2 Dictionary

In sequences, an integer representing a position is associated to each element. A dictionary is a data structure that allows to associate keys of any type to each element.

For example, in real life a dictionary associates a word to a definition (sentences).

1.2.1 Definition

Dictionaries are defined by providing an ensemble of pairs `key:value` separated by comas in-between (curly) brackets: `{key1:val1, ..., keyN:valN}`.

Example :

```
>>> d = {"a":1,"b":2,"c":3}
>>> d = {1:"a",2:"b"}
>>> d = {1:"a","b":2}
```

Remark

The keys and values can be of different types.

Empty Dictionary The empty dictionary is defined by an opening bracket directly followed by a closing one: `{}`.

1.2.2 Accessing Elements

Elements in a dictionary `d` are simply accessed using their associated key value `key` in between square brackets: `d[key]`.

Example :

```
>>> d = {"a":1,"b":2,"c":3}
>>> d["a"]
1
>>> d["b"]
2
```

Warning

If the key you try to access does not exist, Python will throw an error:

```
>>> d = {"a":1,"b":2,"c":3}
>>> d["toto"]
KeyError: 'toto'
```

1.2.3 Adding New Elements

To add a new element to a dictionary `d`, the syntax consist in assigning the value to the corresponding key: `d[key] = value`.

```
>>> d = {}
>>> d[1] = True
>>> d
{1: True}
>>> d[1]
True
```

Chapter 2

Manipulating Sequences

2.1 Slicing

In addition to accessing single elements, Python also allows to access a sub-sequence between two indices called slicing.

Definition 3 *Given two integers* `i < j` *and a sequence* `s`*, the syntax* `s[i:j]` *creates the sequence* `s[i],s[i+1], ..., s[j-1]`*.*

2.1.1 Example

```
>>> l = [1,2,3,"a",None]
>>> l[1:2]
[2]
>>> l[0:3]
[1,2,3]
>>> l[2:5]
[3,"a",None]
>>> "ABCDEF"[1:4]
"BCD"
```

Warning

Be careful when slicing, for `s[i:j]` the last element of the sub-sequence is `s[j-1]` **and not** `s[j]`.

2.1.2 Shortcuts

There exists two specific shortcuts allowed in the slicing syntax, that allow to quickly select every elements up to some position or starting at some position up to the end.

- `s[:j]` is equivalent to `s[0:j]`;
- `s[i:]` is equivalent to `s[i:len(s)-1]`.

2.2 `for` loop

The `for` loop provides a simpler way to perform a sequence traversal (go through all elements) than with a `while` loop. The classical code for going through a sequence `s` with a `while` loop is the following:

```
cpt = 0
while cpt < len(s):
    #Do something with s[cpt]
    cpt = cpt + 1
```

there we have to define a counter variable `cpt`, put the condition `cpt < len(s)` and **not forget** to increment the value of `cpt` at each loop turn. In classical sequence traversal all these steps are **always the same**.

The `for` loop abstracts away all these steps and provides an elegant way to do a sequence traversal. To traverse a sequence `s` using a `for` loop the syntax becomes:

```
for varName in s:
    #Instruction Group
    #that does something with e
```

Where `varname` is a variable name that you can choose. At each loop turn the next value in the sequence `s` is assigned to the variable `varName`.

Remark

Using `e` as variable name, the above for loop is **strictly equivalent** to the following `while` loop:

```
i = 0
while i < len(s):
    e = s[i]
    #Do something with e
    i = i+1
```

2.2.1 Example

```
for e in [1,2,3,None,"a"]
    print(e)
```

Produces the following output:

```
1
2
3
None
a
```

Remark

I strongly advise you to use `for` loops instead of `while` loops when you need to go through a sequence.

Warning

Using this type of `for` loop, you cannot modify the elements of the sequence. The following code **do not** work:

```
>>> l = [1,2,3,4]
>>> for e in l:
        e = e * 2
>>> l
[1,2,3,4]
```

2.3 Range Function

The range function allows to generate list of integers. It is very useful in combination with a `for` where it is used to generate some positions in a sequence in order to mimick back a `while` loop.

2.3.1 The Three Ways Of Using `range`

There exists three possible ways to use the `range` function:

1. `range(n: int) -> list` where `n` is an integer generates the list `[0, 1, ..., n-1]`:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

2. `range(m: int, n: int) -> list` where `m,n` are two integers and `m < n` generates the list `[m,m+1,...,n-1]`:

```
>>> range(3,10)
[3, 4, 5, 6, 7, 8, 9]
```

3. `range(m: int, n: int, s: int) -> list` where `m,n,s` are three integers and `m < n` generates the list `[m,m+s,m+2*s, ..., m+k*s]` where `m+k*s` is the greatest value less than `n`:

```
>>> range(3,10,2)
[3,5,7,9]
>>> range(0,8,2)
[0,2,4,6]
```

2.3.2 range and for

We can use the list of positions returned by the `range` function as the list used in a `for` to simulate the behavior of a `while` loop. With `s` a sequence:

```
for i in range(len(s)):
    #Do something with s[i]
```

is equivalent to:

```
i = 0
while i < len(s):
    #Do something with s[i]
    i = i + 1
```

Warning

In a `for` + `range` loop over a sequence `s`, the elements of `s` are not assigned to a temporary variable as in the `for` loop. Instead elements are accessed directly by specifying their position allowing the content of the list to be modified.

There exists multiple advantages of using a `for` + `range` loop over a `while` loop:

1. First, it requires two less lines of code.
2. Second, as we access the items via their position, it is possible to modify their values in the sequence:

```
>>> l = [1,2,3,4]
>>> for i in range(len(l)):
        l[i] = l[i] * 2
>>> l
[2,4,6,8]
```

3. Finally, we can easily go through a sequence using different steps or starting positions:

```
>>> s = "ABCDEFGHIJK"
>>> for i in range(3, len(s), 3):
        print(s[i])
```

Produces the following output:

```
D
G
J
```