Chapter 11

Slice, Split, Join

It slices, it dices, it even purées!
— never actually said by Ron Popiel

Indexing is a standard feature of most programming languages. Java, C#, C++, and C all have syntax like \( E[i] \) whereby an item with index \( i \) can be reference in \( E \) (which might be a character string, an array of numbers, etc). Python goes further: it is possible to refer to “chunks” of a sequence instead of indexing items one at a time. This feature is called slicing a sequence. Other techniques, like split and join, make it easy to take apart strings and assemble lists of strings into new strings.

In the same way that hand tools allow people to cut, shape, and put together wood into furniture, slicing, split, and join operations form the “carpentry” of Python’s text processing tasks. Even more powerful methods, notably regular expressions, could be likened to the power tools of text and string processing. This chapter only covers the basic techniques, leaving regular expressions and other more advance libraries of software as later topics to explore.

Working with slices, particularly Python’s syntax for slices, takes some getting used to. With practice, it becomes easy to write expressions that take apart sequences in creative ways and use operators to assemble slices into new sequences. For strings, slicing is only the beginning. String methods provide ways to search, replace, and form new strings from sequences of strings.

Slices

A portion of a sequence is called a slice in Python. It’s helpful to visualize a slice in a sequence and name some qualities of the slice.

\[
[9, 8, 7, 9, 2, 3, 8, 6, 5]
\]

start end
Above, the shaded box marks a slice of the sequence and arrows label the places where the slice starts and ends. The Python notation for this slice is

\[9, 8, 7, 9, 2, 3, 8, 6, 5][2:5]\n
This expression looks a bit confusing due to the square brackets. It’s easier on the eyes to see a similar string slice:

```python
>>> "wonderful"[2:5]
'nde'
```

The notation for slicing is like indexing, but uses a colon to separate the start and end of the slice: `[start,end]` specifies the slice to Python. The `start` index comes before the colon; the `end`, however, is not the index of the ending item — instead `end` is just beyond the index of the last item of the slice. Initially, this might seem like bad language design. Why not be consistent, why not use an index value for both start and end positions of the slice? To answer this (and also give students and easy way to remember how slicing works) here is a rewriting of the previous expression:

```python
>>> "wonderful"[2:2+3]
'nde'
```

Of course, `2+3` is just 5. But the point is that we look at `2+3` here as being the `start` index (2) plus the length of the slice (3 characters). Seen this way, the notation makes sense. The length of the slice is `end - start`.

**Slices are Sequences** One deceptive behavior of slicing is a slice of length 1:

```python
def slicomp(seq,start,end):
    print ( seq[start:end], seq[start] )
    print ( type( seq[start:end] ), type( seq[start] ) )
    return seq[start:end] == seq[start]
...
>>> slicomp('tambor',3,4)
b b
<class 'str'> <class 'str'>
True
>>> slicomp( [True,7,False,9,True,5], 3,4)
[9] 9
<class 'list'> <class 'int'>
False
```
What this example reveals is that indexing works slightly differently for strings than it does for lists or tuples. On strings, indexing returns a string, whereas on the other types, indexing returns an element of the list or tuple. Notice that Python finds [9] and 9 have different types and therefore, are not equal (they are not the same thing). This fact is obvious for slices of length 2 or more, but it’s easy to forget this with a slice of length 1. The simple rule to remember is that a slice is always a sequence, even if its length is 1.

**Maximum Slices**  Another special case for a slice is shown here:

```python
def reprod(seq):
    print seq, seq[0:len(seq)]
...
>>> reprod("deals")
deals deals
>>> reprod([9,8,7])
[9, 8, 7] [9, 8, 7]
>>> reprod((False,True))
(False, True) (False, True)
```

The example shows what happens when a slice starts with 0, the index of the first item in the sequence, and ends just after the index of the last item, that is, \( \text{len(seq)} \). The value of \( \text{len(seq)} \) is not a valid index value – it lies outside of the sequence; however, it is meaningful for slicing. In fact, slicing notation is more forgiving that indexing:

```python
>>> "abc"[0:999]
'abc'
```

Any value beyond the index of the sequence’s last item works to get a slice including the rightmost item of the sequence. Of course, if a large value is used for the end, that will violate the earlier “+2” observation:

```python
>>> "abc"[1:1+99]
'bc'
```

Python cannot give back a slice of length 99 when the original sequence, starting at index 1, only has two items. Thus Python returns the longest slice possible with the specified starting index value.

**Empty Slices**  Many expressions for slicing result in empty sequences: ‘hello’ [0:0], ‘hello’ [1:1], ‘hello’ [2:2], ‘hello’ [3:3], and ‘hello’ [4:4] are all empty.
slices. Visually, these examples correspond to putting the end so close to the start that there’s no room for any item, shown here for start/end [2:2]

\[ [ 9, 8, 7, 9, 2, 3, 8, 6, 5 ] \]

Again, Python is rather tolerant about what can be put as start and end of a slice that is empty; even 'hello'[999:999] will produce an empty slice. Also, if the end is smaller than the start value, 'hello'[3:0], that is an empty slice. A takeaway from this is that sequences contain lots of empty slices, so to speak.

**Negative Index Values** Just as some negative numbers work as index values (-1 for the end of a sequence), you may use negative numbers to specify both start and end of a slice. For instance, here is an expression to get a slice starting with the second character and ending just before the end of a string.

```python
>>> "imagine"[1:-1]
'magin'
```

**Default Values** The syntax for slicing has some handy ways to say that a sequence starts with index 0 or that it includes everything up to the last item. If you omit the start value, the default is 0; if you omit the end value, the default is to include the rest of the sequence:

```python
>>> "tertiary junction"[10:]
'unction'
>>> "anthropology"[:8]
'anthropo'
>>> 'anthropology'[:8]
'anth'
>>> "empire"[:]
'empire'
```

**Split**

The **split** method works for strings only, not for other species of sequence. The idea is to split up an input string into multiple strings, where the splitting points are determined by a pattern. Date formats typically have slash characters separating the date fields. Using **split** the fields of a date are extracted.

```python
'10/25/2006'.split('/
)')
['10', '25', '2006']
```
Notice that the pattern, the ‘/’ character in this case, is not present in the output: the splitting pattern is removed. Also, the output is not a string, but a list of strings. There are several natural questions one might have about the `split` method:

- What happens if the pattern isn’t found in the input?
- What happens if the input is nothing but the pattern?
- Can the pattern be anything? What about a pattern with more than a single character? How about an empty string?
- Is there some way to have more than one pattern?

These questions are answered later by showing examples. The point of listing the questions is that a single example is rarely enough to fully understand an operator, function or method. How a method behaves with exceptional inputs, like corner cases, need to be investigated for full understanding. Good reference manuals for programming languages (not textbooks) should explain all the details of functions and methods, answering all questions about what happens with unusual inputs. One problem in practice is that even when reference manuals do explain details, the explanations could be mathematical or use “legalese”, that is, English found in laws, regulations, or bureaucratic text that is hard to fathom. Fortunately, you can always try examples with Python and discover for yourself the answers.

More generally, the problem of clearly describing how software works or should behave is a profound, core challenge in computing. Misunderstandings by users, by programmers, and by software vendors can have serious consequences. In this introductory text on computing we won’t see much that helps to resolve such problems; this is a more advanced topic. Some years ago, IBM developed a new computer and new programming language and wanted to test the new system. The company chose a novel strategy. They loaned some early models of the new system to local schools, where kids could play with them. What IBM found is that the system quickly broke, both hardware and software, and it broke in many different ways. It seems that kids tried doing things that the programmers and engineers had never imagined. Probably, had the system only been sold to other companies with business applications, the bugs associated with unanticipated inputs and surprising usage patterns might have taken years to be exposed, if ever. For high quality software, the old saying “expect the unexpected” is good advice to software designers. On a small scale, that’s another reason to explore the corner cases of inputs to methods and functions.

```python
>>> 'testing 123'.split('/')
['testing 123']
```

This shows what happens when the splitting pattern is absent from the input string: the result is a list, but it only contains the original input string.
>>> "on knowing the price of dice".split('e ')
['on knowing th', 'pric', 'of dice']

Yes, you can use a pattern that is a string longer than a single character. But, it can be confusing:

>>> "oooo ooooo oooooo".split('oo')
['', ' ', ' ', ' ', 'o ', ' ', ' ', ' ']

To make sense of the result, it may help to put lines under and over the places where the pattern occurs (we use both under and over to make it easy to see).

```
\underline{oooo} \underline{oooo} \underline{oooo}
```

Between \underline{oo} and \underline{oo} in \underline{ooooo}, there is an empty string (see it?). That’s the rationale for some of the empty strings in the output. All the overlined and underlined portions, which match the pattern, are removed. The resulting list only contains strings that are between the removed patterns, or between a removed pattern a the start or the end of the original string.

```python
>>> "oo".split('oo')
['', ' ']
```

Another way to interpret the above result is that, technically,

"oo" is obtained by evaluating \( \text{'' + "oo" + ''} \)

and removing the pattern 'oo' leaves an empty string representing the start and another representing the end of the original string. By rewriting a string containing the pattern as a concatenation expression (\(*\)), the output result makes some sense.

The `split` method doesn’t work for just any pattern: the pattern cannot be empty, or Python will output an error message.

```python
>>> "abc".split('')
ValueError: empty separator
```

**Whitespace Split**

One other special case for a pattern is for splitting by *whitespace*. Whitespace refers not to a single string, but any combination of characters that are invisible when printed, such as blanks, tabs, newlines, and such. The aim of whitespace splitting is to derive a list of the words in an input string, without having any empty strings, blank strings, or similar, in the list. The whitespace split is signalled to Python by have *no argument* to the pattern. Thus,
Slice, Split, Join

>>> "One\n\ntTwo Three Four\n\nFive\n".split()
['One', 'Two', 'Tree', 'Four', 'Five']

The whitespace split is especially useful for functions that might get input text lines in either Unix/Linux format or in Windows format: the usual way lines end in Windows is with the string '\r\n', whereas in other systems lines end with just '\n'. The whitespace split produces the same result with either convention for ending a line of text.

Join

The `join` method does the opposite of `split`, roughly speaking.

```python
>>> ' '.join(['how','big','is','the','storm'])
'how big is the storm'
>>> '/'.join(['12','31','15'])
'12/31/15'
>>> '--'.join(['a','b','c','d'])
'a--b--c--d'
```

The `join` method thus takes a *separator string* and uses that as the “glue” when joining up (actually, concatenating) a list of strings. Suppose there is a list of strings to be concatenated all together; then use the empty string for the separator, and `join` does the work:

```python
>>> ''.join(['how','big','is','the','storm'])
'howbigisthestorm'
```

Other String Methods

Many other string methods are built-in to Python and yet more are available in the Python libraries of functions and methods. A few are shown here with examples.

```python
>>> 'round Table'.upper()
'ROUND TABLE'
>>> 'round Table'.lower()
'round table'
>>> 'Inventory'.center(30)
   Inventory
>>> "mississippi".replace('i','x')
'msssxssxppx'
```
The `center` method produces an output string of the specified length, roughly centering the input string into the output. The `strip` method removes whitespace to the left and to the right of any non-whitespace text in the input. There are also `lstrip` and `rstrip` methods that, respectively, remove only from the left or right. The `count` method reports how many times the method’s argument occurs in the string. The `count` method is notable in that it also works for lists:

```python
>>> [6>5, 5>4, 1>2, 'e'>'g'].count(True)
2
```

## Method on Method

Though Chapter 6 explains how an expression can use multiple function applications and method calls, there are some tricks of the trade you might not think of initially using Python. It’s helpful to see some examples.

```python
>>> ''.join("1/2/3".split('/')) == "123"
True
```

The first example shows that expressions with methods can be arguments to other methods. The next example deals with an unanswered question posed earlier in the chapter. How can we split on multiple patterns, that is, do something like the whitespace split, but for some custom patterns? Below is Python code to split either using a colon or a semicolon.

```python
>>> 'e;f or g:t m;d'.replace(':', ';').split(';;')
['e', 'f or g', 't m', 'd']
```
The example exploits the way that Python evaluates expressions left to right, with exceptions for operator priorities. The `replace` method is the first to be evaluated, and it creates a string with colons changed to semicolons; then the `split` method divides the string into a list of strings according to where semicolons were found. The same idea is useful to compare an input string without caring about the difference between upper or lower case:

```python
def anycompare(S,T):
    return S.upper() == T.upper()
...
>>> anycompare("Fast","Fast")
True
>>> anycompare("slow","slowW")
False
```

**Terminology Review**

Terminology introduced in this chapter includes: slice, whitespace, join, separator string, and replace.

**Exercises**

(1) What does Python evaluate the following expressions to be?

2. [115,202,192,334,257][:4]
3. len("crazy"[3:3+4])
4. [9,8,7,6,5,4,3,2,1][-3:]
5. type([False,True,False,True][2:3])
6. "---".join( "this is important".split() )
7. int( "7/7/07".split( '/' ) )
8. "too soon to tell".replace( 'o','*' ).replace( ' '*,'' )

(2) What does the following script print?

```python
def midcap(f):
    return f[:f.index("t")].split()[2]
print f("going far in a boat over a river")
```

(3) Use a search engine to look up the Python `endswith` method and the Python `rstrip` method. Then write a function `ing(w)` that returns `True` if `w` is a string that ends with "ing", but ignoring any whitespace.
>>> ing("barely")
False
>>> ing("run or walking ")
True
>>> ing("Douglas and Boeing")
True