Programming Fundamentals

Data Types: Strings

João Correia Lopes

INESC TEC, FEUP

25 October 2018

Goals

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

- Describe how to work with strings as single things
- Describe how to work with the parts of a string
- Enumerate the main methods available to work with strings
- Describe how to format strings

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Section 5.1) [PDF]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Chapter 6) [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 8) [HTML]

Tips

- There's no slides: we use a script and some illustrations in the class. That is NOT a replacement for **reading 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 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 and pairs
- Strings, lists, and pairs are qualitatively different from the others because they are made up of smaller pieces
- In the case of strings, they're made up of smaller strings each containing one **character** in a particular order from left to right
- 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

WORKING WITH STRINGS AS SINGLE THINGS

- Just like a turtle, a string is also an object
- So each string instance has its own attributes and methods (around 70!)
- For example:

```
>>> our_string = "Hello, World!"
>>> all_caps = our_string.upper()
>>> all_caps
'HELLO, WORLD!'
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/smethods.py

WORKING WITH THE PARTS OF A STRING

- The **indexing operator** selects a single character substring from a string:

```
>>> fruit = "banana"    # a string
>>> letter = fruit[0]   # this is also a string
>>> print(letter)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/sindex.py

LENGTH

- The `len` function, when applied to a string, returns the number of characters in a string:

```
1  size = len(word)
2  last = word[size-1]
```

TRAVERSAL AND THE FOR LOOP

- A lot of computations involve processing a *string one character at a time*
- Often they start at the beginning, select each character in turn, do something to it, and continue until the end
- This pattern of processing is called a **traversal**

```
1  word = "Banana"
2  for letter in word:
3      print(letter)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/straversal.py

SLICES

- A substring of a string is obtained by taking a slice
- Similarly, we can slice a list to refer to some sublist of the items in the list

⇒ https://github.com/fpro-admin/lectures/blob/master/10/slices.py

STRING COMPARISON

- The comparison operators work on strings
- To see if two strings are equal:

```
1  if word == "banana":
2      print("Yes, we have bananas!")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/scomparison.py

STRINGS ARE IMMUTABLE

- Strings are **immutable**, which means you can't change an existing string
- The best you can do is create a new string that is a variation on the original

```
1  greeting = "Hello, world!"
2  greeting[0] = 'J'             # ERROR!
3
4  greeting = "J" + greeting[1:]
5  print(greeting)
```

THE IN AND NOT IN OPERATORS

- The `in` operator tests for membership
- The `not in` operator returns the logical opposite results of `in`

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "apple" in "apple"
True
>>> "" in "a"
True
>>> "x" not in "apple"
True
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/removeVowels.py

A FIND FUNCTION

- In a sense, find is the opposite of the indexing operator
- What does the following function do?

```
1  def my_find(haystack, needle):
2  """
3  Find and return the index of needle in haystack.
4  Return -1 if needle does not occur in haystack.
5  """
6  for index, letter in enumerate(haystack):
7      if letter == needle:
8          return index
9  return -1
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/sfind.py

LOOPING AND COUNTING

- another example of the **counter** pattern introduced in *Counting digits*

```
1  def count_a(text):
2      count = 0
3      for letter in text:
4          if letter == "a":
5              count += 1
6      return count
7
8  print(count_a("banana") == 3)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/scount.py

OPTIONAL PARAMETERS

- To find the locations of the second or third occurrence of a character in a string
- we can modify the `find` function, adding a third parameter for the starting position in the search string
- Better still, we can combine find and find2 using an **optional parameter**:

```
1  def find(haystack, needle, start=0):
2      for index,letter in enumerate(haystack[start:]):
3          if letter == needle:
4              return index + start
5      return -1
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/optional.py

THE BUILT-IN FIND METHOD

- The built-in `find` method is more general
- It can find substrings, not just single characters:

```
1  >>> "banana".find("nan")
2  2
3  >>> "banana".find("na", 3)
4  4
```

THE SPLIT METHOD

- One of the most useful methods on strings is the split method
- it splits a single multi-word string into a list of individual words, removing all the whitespace between them[1]

```
>>> phrase = "Oh, that's jolly good. Well, off you go then"
>>> words = phrase.split()
>>> words
['Oh,', "that's", 'jolly', 'good.', 'Well,', 'off', 'you', 'go', 'then']
```

[1] Whitespace means any tabs, newlines, or spaces.

CLEANING UP YOUR STRINGS

- We'll often work with strings that contain punctuation, or tab and newline characters
- But if we're writing a program, say, to count word frequency, we'd prefer to strip off these unwanted characters.
- We'll show just one example of how to strip punctuation from a string
  - we need to traverse the original string and create a new string, omitting any punctuation

⇒ https://github.com/fpro-admin/lectures/blob/master/10/strip.py

THE STRING FORMAT METHOD

- The easiest and most powerful way to format a string in Python3 is to use the `format` method
- The template string contains place holders, ... `{0}` ... `{1}` ... `{2}` ... etc
- The format method substitutes its arguments into the place holders
- To see how this works, let's start with a few examples:

```
1  phrase = "His name is {0}!".format("Arthur")
2  print(phrase)
3
4  name = "Alice"
5  age = 10
6  phrase = "I am {0} and I am {1} years old.".format(age, name)
7  print(phrase)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/10/format.py

FORMAT SPECIFICATION

- Each of the replacement fields can also contain a **format specification**
- This modifies how the substitutions are made into the template, and can control things like:
  - whether the field is aligned to the left `<`, center `^`, or right `>`
  - the width allocated to the field within the result string (a number like 10)
  - the type of conversion: we'll initially only force conversion to float, `f` or perhaps we'll ask integer numbers to be converted to hexadecimal using `x`)
  - if the type conversion is a float, you can also specify how many decimal places are wanted: typically, `.2f` is useful for working with currencies to two decimal places

⇒ https://github.com/fpro-admin/lectures/blob/master/10/formatspec.py

EXERCISES

- Moodle activity at: LE10: Strings