M30299 – Programming
Lecture 06 – Computing with Strings

Matthew Poole & Nadim Bakhshov
`moodle.port.ac.uk`

School of Computing
University of Portsmouth

2020/21

Introduction to lecture

- In this lecture we introduce Python's **string** (or `str`) data type.
- As we will see, strings are kinds of **sequences**; other sequences include lists, which we will see here and cover in detail later.

The string data type

- We have used many string values in the first few practicals.
- Let's see a couple of examples:

```
>>> name = "Sam"
>>> greeting = 'Hello'
>>> name
'Sam'
>>> print(name)
Sam
>>> type(name)
<class 'str'>
>>> type(greeting)
<class 'str'>
```

String operations

- Like the numerical data types, the string type has some **operators** associated with it; including:
  - + (concatenation), and
  - * (repetition).

```
>>> greeting + "There"
'HelloThere'
>>> name * 3
'SamSamSam'
```

- The function `len` gives the number of characters in a string:

```
>>> len(greeting)
5
```

String indexing

- A string is just a **sequence** of characters. The **positions** of a string can be numbered (using ints), starting with 0, as follows:

0	1	2	3	4	5	6	7
'H'	'i'	' '	't'	'h'	'e'	'r'	'e'

- We can **access** individual characters of a string using the **indexing** notation `string[position]`. For example:

```
>>> greeting = "Hi there"
>>> greeting[3]
't'
```

String indexing

```
>>> greeting = "Hi there"
>>> i = 4
>>> greeting[i+2]
'r'
```

- Python strings can also be indexed using **negative indices**, where -1 is the position of the final character:

```
>>> greeting[-1]
'e'
>>> greeting[-4]
'h'
```

String slicing

- As well as accessing individual characters, we can also access **substrings** using an operation called **slicing**.
- To do this, we use the notation `string[start:end]`.
- This will give the substring starting at position `start`, and ending one position short of `end`. For example:

```
>>> greeting = "Hi there"
>>> greeting[0:2]
'Hi'
>>> greeting[3:6]
'the'
>>> greeting[3:]
'there'
```

Strings, lists and sequences

- In Practical Worksheet P1, we saw an example of a **list**:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

and a **loop**:

```
>>> for i in range(10):
        print(i, end=" ")


0 1 2 3 4 5 6 7 8 9
```

- (Note: the `end=" "` above tells the `print` function to display a space after printing, rather than a newline.)

Strings, lists and sequences

- Strings and lists are both examples of **sequences**, and as such, they share many properties.
- For example, we can use a loop with a string:

```
>>> for ch in "Sam":
        print(ch)


S
a
m
```

Strings, lists and sequences

- We can also concatenate, index, and slice lists; for example:

```
>>> myList = [3, 2, 7, 1]
>>> myList + [3, 4]
[3, 2, 7, 1, 3, 4]
>>> len(myList)
4
>>> myList[2]
7
>>> myList[1:3]
[2, 7]
```

String methods

- There are many other operations on strings. These operations take the form of **methods**.
- There are several useful string methods; we'll look at a few:

```
>>> myString = "How are you today"
>>> myString.upper()
'HOW ARE YOU TODAY'
>>> myString.replace("are", "were")
'How were you today'
>>> myString.count('a')
2
```

String methods

- One of the most useful methods is `split`, which splits a string into a list of words (or substrings):

```
>>> myString.split()
['How', 'are', 'you', 'today']
```

- An example use of this is a word counter function:

```
def wordCounter():
    line = input("Enter a line of text: ")
    wordList = line.split()
    print("You entered", len(wordList), "words")
```

String formatting

- Sometimes programs need to display nicely formatted output; e.g:
  - display a float to two decimal places; or
  - display a column of numbers that are right justified.
- To do this, we use Python's string `format` method; we'll just look at a few examples.
- We first need to understand how **placeholders** or **slots** work:

```
>>> e = 10.6
>>> p = 2
>>> "Pay {0} euros for {1} pizzas".format(e, p)
"Pay 10.6 euros for 2 pizzas"
```

String formatting

- The slots {0} and {1} are filled by the **arguments** of format.
- Here, e is argument 0, and p is argument 1.
- Let's add a **format specifier** for slot number 0:

```
>>> "Pay {0:7.2f} euros for {1} pizzas".format(e, p)
"Pay   10.60 euros for 2 pizzas"
```

- In the format specifier 7.2f
  - the .2f means use 2 decimal places.
  - the 7 tells Python to use 7 characters in total (five are needed for 10.60 and two extra are added at the beginning).

String formatting - examples

```
>>> "Pay {0:0.3f} euros for {1} pizzas".format(e, p)
'Pay 10.600 euros for 2 pizzas'
```

- Here, since the 0 in `0.3f` specifies insufficient space, Python ignores it and uses as many characters (here 6) as necessary.

```
>>> "Pay {0:0.0f} euros for {1} pizzas".format(e, p)
'Pay 11 euros for 2 pizzas'
```

- Here, we remove all decimal places (note the rounding).

```
>>> "Pay {0} euros for {1:0.1f} pizzas".format(e, p)
'Pay 10.6 euros for 2.0 pizzas'
```

String formatting - examples

- The above format specifiers cause values to be right justified.
- Often, especially when formatting text, we wish to to use left and centre justification.
- We use < (left), > (right) and ^ (centre):

```
>>> word = "HELLO!"
>>> print("A big {0:<12} to you!".format(word))
A big HELLO!       to you!
>>> print("A big {0:>12} to you!".format(word))
A big       HELLO! to you!
>>> print("A big {0:^12} to you!".format(word))
A big    HELLO!    to you!
```