Lecture 4.1 – Computing with Strings

M30299 Programming

School of Computing
University of Portsmouth

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).
- For example,

```
>>> 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 integers), starting with 0, as illustrated by:

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 1, we saw an example of a **list**:

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

and a **loop**:

```
>>> for i in range(5):
        print(i, end=" ")

0 1 2 3 4
```

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

Strings, lists and sequences

- Strings, ranges and lists are both examples of **sequences**, and as such, they share many properties.
- For example, we can use a loop to go through the characters of 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:

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

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

String methods

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

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

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

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

String formatting

- Sometimes programs need to display nicely formatted output; for example, to:
    - display a float to two decimal places; or
    - display a column of numbers that are right justified.
- To do this, we can use Python's **f-strings**; we'll just look at a few examples.
- An f-string is like a normal string value but prefixed by an `f`. Inside the string you can put expressions within `{}`.
- These expressions are evaluated to give a normal string:

```
>>> price = 10.6
>>> topping = "cheese and tomato"
>>> print(f"Pay {price} euros for a {topping} pizza.")
Pay 10.6 euros for a cheese and tomato pizza.
```

String formatting

- We can add a **format specifier** to achieve a nice format for the cost:

```
>>> print(f"Pay {price:.2f} euros for a {topping} pizza.")
Pay 10.60 euros for a cheese and tomato pizza.
```

- Here, the `.2f` means 2 decimal places.
- We can add another number immediately after the `:` to specify the minimum number of characters that should appear in the string:

```
>>> print(f"Pay {price:6.2f} euros for a {topping:20} pizza.")
Pay  10.60 euros for a cheese and tomato    pizza.
```

- Here,
  - one padding space has been added before the price to give 6 characters in total;
  - three paces have been added after the topping to give 20 characters in total.