University of Huddersfield

CFS2160: Software Design and Development

03: Strings and Tuples

More Data Types. Objects and Classes.

Tony Jenkins
A.Jenkins@hud.ac.uk

```python
name = input ('Enter the student\'s name: ')

mark_1 = int (input ('Enter first result:  '))
mark_2 = int (input ('Enter second result: '))
mark_3 = int (input ('Enter third result:  '))
mark_4 = int (input ('Enter fourth result: '))
mark_5 = int (input ('Enter fifth result:  '))

total_marks = mark_1 + mark_2 + mark_3 + mark_4 + mark_5

average_mark = total_marks / 5

print ()
print ('Final Mark for ' + name + ' is ' + str (average_mark))
```

Strings

Any sequence of characters is a *string*.

In programs, strings appear between quotation marks:

- ➢ Use `"` or `'` (it doesn't matter which, but be consistent.)

This means that:

- ➢ `"1"` is a string containing the character `1`.
- ➢ `1` is the integer value, one.
- ➢ `'True'` is a string containing four characters `T r u e`.
- ➢ `True` is the Boolean truth value.

Strings

Any ***sequence*** of characters is a *string*.

In programs, strings appear between quotation marks:

- ➢ Use `"` or `'` (it doesn't matter which, but be consistent.)

This means that:

- ➢ `"1"` is a string containing the characte
- ➢ `1` is the integer value, one.
- ➢ `'True'` is a string containing four cha
- ➢ `True` is the Boolean truth value.

We say a *sequence* because the order of characters in a string is (usually) important.

Indexing

If the order matters, it follows that we need a way to reference individual characters.

This is *indexing*: each character in the string has an index.

Indexes start at zero, and a square bracket contains the index:

- `knight = "Brave Sir Robin"`
  - `knight [0]` is `'B'`
  - `knight [2]` is `'a'`
  - and so on.

More Indexing

This is fine, but what if we want the *last* character (which we often do)?

In some languages, we would find the length, subtract one, and look at that index.

Python does it better! Negative index values count from the end of the string:

- `knight = "Brave Sir Robin"`
  - `knight [-1]` is `'n'`
  - `knight [-3]` is `'b'`
  - and so on.

Slices

What if we want to extract some characters?

In some languages, this is difficult, and has to be coded by hand.

Python does it better! A *slice* extracts a substring from the string:

- ➢ `knight = "Brave Sir Robin"`
  - ➢ `knight [0:5]` is `'Brave'`
  - ➢ `knight [:5]` is also `'Brave'`
  - ➢ `knight [6:]` is `'Sir Robin'`
  - ➢ `knight [::2]` is 'BaeSrRbn'

Slices

What if we want to extract some characters?

In some languages, this is difficult, and has to be coded by hand.

Python does it better! A *slice* extracts a substring from the string:

- `knight = "Brave Sir Robin"`
  - `knight [0:5]` is `'Brave'`
  - `knight [:5]` is also `'Brave'`
  - `knight [6:]` is `'Sir Robin'`
  - `knight [::2]` is 'BaeSrRbn'

It's `[start_index : stop_index : step]`.

Strings are Objects

A string is an object, and like all objects a string has:

- ➢ Some data values.
- ➢ Some operations that manipulate these values.

The data values are the characters in the string.

We have seen two operations: indexing and slicing.

There are many more, defined as *functions*.

Strings are Objects

A string is an object, and like all objects a string has:

- ➢ Some data values.
- ➢ Some operations that manipulate these values.

The data values are the characters in the string.

We have seen two operations: indexing and slici

There are many more, defined as *functions*.

This is a good time to point out the Python docs.

All the details of strings are at:

https://docs.python.org/3.6/library/string.html

Strings are Objects

A string is an object, and like all objects a string has:

- ➢ Some data values.
- ➢ Some operations that manipulate these values.

The data values are the characters in the string.

We have seen two operations: indexing and slici

There are many more, defined as *functions*.

This is a good time to point out the Python docs.

Always check the version!

https://docs.python.org/**3.6**/library/string.html

String Functions: length

Suppose we want to know how long a string is (or even just if it is not empty).

The function `len` does that. It is a *function*, so is used like so:

- ➢ `knight = ""` `len (knight)` is zero.
- ➢ `knight = 'Brave Sir Robin'` `len (knight)` is 15.

String Functions: length

Suppose we want to know how long a string is (or even just if it is not empty).

The function `len` does that. It is a *function*, so is used like so:

- ➢ `knight = ""` `len (knight)` is zero.
- ➢ `knight = 'Brave Sir Robin'` `len (knight)` is 15.

`len` is "interesting" because it can be used on any object that has a length.

- ➢ The number of items in a list, say.

Other functions are specific to strings, so are used slightly differently.

String Functions: length

Suppose we want to know how long a string is (or even just if it is not empty).

The function `len` does that. It is a *function*, so is used like so:

- ➢ `knight = ""`   `len (knight)` is zero.
- ➢ `knight = 'Brave Sir Robin'`   `len (knight)` is 15.

`len` is "interesting" because it can be used on any object that has a length.

- ➢ The number of items in a list, say.

Other ~~functions~~ methods are specific to strings, so are used slightly differently.

String Methods: Converting Case

It is common to want to convert the case of characters in a string.

```
>>> s = 'ni! NI!'

>>> s.lower ()
'ni! ni!'

>>> s.upper ()
'NI! NI!'

>>> s.swapcase ()
'NI! ni!'

>>> s.capitalize ()
'Ni! ni!'
```

String Methods: Converting Case

It is common to want to convert the case of characters in a string.

These operations are common, so they are part of the Python Standard Library.

Whenever writing a program, always look through the library to find stuff that will help!

Do not reinvent the wheel!

```
>>> s = 'ni! NI!'

>>> s.lower ()
'ni! ni!'

>>> s.upper ()
'NI! NI!'

>>> s.swapcase ()
'NI! ni!'

>>> s.capitalize ()
'Ni! ni!'
```

String Methods: Converting Case

It is common to want to convert the case of characters in a string.

These operations are common, so they are part of the Python Standard Library.

Whenever writing a program, always look through the library to find stuff that will help!

Do not reinvent the wheel!

And if it's not in the Standard Library, the chances are someone else has already done it. See the Python Package Index (PyPI):

https://pypi.python.org/pypi

```
>>> s = 'ni! NI!'

>>> s.lower ()
'ni! ni!'

>>> s.upper ()
'NI! NI!'

>>> s.swapcase ()
'NI! ni!'

>>> s.capitalize ()
'Ni! ni!'
```

String Methods: Finding Substrings

Often we want to find a substring within a string.

```
>>> k = 'Brave Sir Robin'
>>> k.find ('Sir')
6

>>> k.rfind ('i')
13

>>> k.find ('Ni!')
-1
```

String Methods: Finding Substrings

Often we want to find a substring within a string.

These methods give the index of the *first occurrence* of the string, searching from left or right.

They are *case sensitive*.

```
>>> k = 'Brave Sir Robin'
>>> k.find ('Sir')
6

>>> k.rfind ('i')
13

>>> k.find ('Ni!')
-1
```

String Methods: Finding Substrings

Often we want to find a substring within a string.

These methods give the index of the *first occurrence* of the string, searching from left or right.

They are *case sensitive*.

```
>>> k = 'Brave Sir Robin'
>>> k.find ('sir')
-1
```

String Methods: Finding Substrings

Often we want to find a substring within a string.

These methods give the index of the *first occurrence* of the string, searching from left or right.

They are *case sensitive*.

So sometimes we *chain* method calls together.

Here we convert the string to lowercase before doing the search (so we match "Sir", "sir" or even "siR".

```
>>> k = 'Brave Sir Robin'
>>> k.find ('sir')
-1

>>> k.lower ().find ('sir')
6
```

String Methods: in

Sometimes it is sufficient to test if a substring is present in the string.

This could be done with `find ()`, and then checking if the returned value was not `-1`.

A shorthand - `in` - exists to do this check.

```
>>> k = 'Brave Sir Robin'

>>> 'sir' in k
False

>>> 'Sir' in k
True

>>> 'sir' in k.lower ()
True
```

String Methods: in

Sometimes it is sufficient to test if a substring is present in the string.

This could be done with `find ()`, and then checking if the returned value was not `-1`.

A shorthand - `in` - exists to do this check.

This works for any object where it makes sense to "test membership": lists, for example.

```
>>> k = 'Brave Sir Robin'

>>> 'sir' in k
False

>>> 'Sir' in k
True

>>> 'sir' in k.lower ()
True
```

More String Methods

```
>>> k = "Brave Sir Robin"

>>> k.count ('i')
2

>>> k.replace ('Brave', 'Timid')
'Timid Sir Robin'
```

More String Methods

```
>>> k = "Brave Sir Robin"

>>> k.count ('i')
2

>>> k.replace ('Brave', 'Timid')
'Timid Sir Robin'
```

Important
Does `replace` change the value of the string in k?
Not sure? Check:

https://docs.python.org/**3.6**/library/string.html

Maths on Strings

Sometimes it makes sense to apply mathematical operators to an object.

For strings, two make sense: + to add two strings, * to repeat a string.

```
➢ >>> k = 'Sir' + ' ' + 'Robin'
➢ >>> shout = 'Ni! ' * 3
```

Division and subtraction make no sense for strings, so are not implemented.

Errors

In all of this, we have ignored the possibility of errors.

What happens, for example, if we try to access an index value that is too big?

```
>>> k = 'Brave Sir Robin'

>>> k [64]
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
IndexError: string index out of
range
```

Errors

In all of this, we have ignored the possibility of errors.

What happens, for example, if we try to access an index value that is too big?

This error is an *Exception*.

As we get experience we will see how to stop these happening.

For the moment, don't worry, and if all else fails:

"Google the Error Message."

```
>>> k = 'Brave Sir Robin'

>>> k [64]
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
IndexError: string index out of
range
```

Collections and Sequences

We know that:

- ➢ A string is a sequence of characters.
- ➢ The order of the characters is important.

To generalise, a string is a *collection* of characters, where the *order* is important.

In Python we can build collections of any types of objects.

In these collections, sometimes the order is important, sometimes it is not.

A Collection

Suppose we want to store the ingredients for a pizza.

We could think of it as a *collection* of *ingredient* objects.

The order of this collection does not matter.

But we will want to search it, and maybe add or delete items.

A Collection

Suppose we want to store the ingredients for a pizza.

We could think of it as a *collection* of *ingredient* objects.

The order of this collection does not matter.

But we will want to search it, and maybe add or

It should be obvious why storing this as a single string would not be the best idea.

A Collection

Suppose we want to store the ingredients for a pizza.

We could think of it as a *collection* of *ingredient* objects.

The order of this collection does not matter.

But we will want to search it, and maybe add or

Data Structure
We are talking about *data structures* here.
Picking the right one is important, and comes with experience.

Tuples

A tuple is a very simple collection.

It is *exactly* the same thing as a string, except:

- ➢ A string has *elements* that are characters.
- ➢ A tuple has *elements* that are any objects.

The elements in a tuple do not have to be all the same type (although they usually are).

A tuple can have another tuple as an element (!).

Tuples

A tuple is denoted by round brackets, and is declared like any other object.

```
>>> knights = ()
>>> type (knights)
<class 'tuple'>
```

Tuples

A tuple is denoted by round brackets, and is declared like any other object.

Elements can be added when the tuple is created.

```
>>> knights = ('Robin', 'Galahad')
```

Tuples

A tuple is denoted by round brackets, and is declared like any other object.

Elements can be added when the tuple is created.

If there is only one initial element, there **must** be a comma after it.

```
>>> knights = ('Robin')
>>> type (knights)
<class 'str'>

>>> knights = ('Robin',)
>>> type (knights)
<class 'tuple'>
```

Tuples

A tuple is denoted by round brackets, and is declared like any other object.

Elements can be added when the tuple is created.

If there is only one initial element, there **must** be a comma after it.

So most programmers include the comma even if there are many elements.

```
>>> knights = ('Robin')
>>> type (knights)
<class 'str'>

>>> knights = ('Robin',)
>>> type (knights)
<class 'tuple'>

>>> knights = ('Robin', 'Galahad',)
>>> type (knights)
<class 'tuple'>
```

Using Tuples

"Tuples are the same as strings".

It follows that many of the same operations will work …

So `len` works as expected.

```
>>> k = ('Robin', 'Arthur', 'Galahad')

>>> len (k)
3
```

Using Tuples

"Tuples are the same as strings".

It follows that many of the same operations will work …

So `len` works as expected.

Indexes and slices work too.

(Note that a slice returns another tuple - a slice of a string returns a string.)

```
>>> k = ('Robin', 'Arthur', 'Galahad')

>>> len (k)
3

>>> k [1]
'Arthur'
>>> k [-1]
'Galahad'
>>> k [::2]
('Robin', 'Galahad')
```

Using Tuples

"Tuples are the same as strings".

It follows that many of the same operations will work …

`in` can test membership.

```
>>> k = ('Robin', 'Arthur', 'Galahad')

>>> 'Bedevere' in k
False
>>> 'Arthur' in k
True
```

Using Tuples

"Tuples are the same as strings".

It follows that many of the same operations will work …

`in` can test membership.

Perhaps surprisingly, `find` does not work, but remember it was a *method* in the string class.

```
>>> k = ('Robin', 'Arthur', 'Galahad')

>>> 'Bedevere' in k
False
>>> 'Arthur' in k
True

>>> k.find ('Robin')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no
attribute 'find'
```

Using Tuples

"Tuples are the same as strings".

```
>>> k = ('Robin', 'Arthur', 'Galahad')
```

It follows that many of the same operations will work …

`in` can test membership.

```
>>> 'Bedevere' in k
False
>>> 'Arthur' in k
True
```

Perhaps surprisingly, `find` does not work, but remember it was a *method* in the string class.

For tuples, `index` does the job.

```
>>> k.index ('Robin')
0
```

Tuples are Limited

Possibly we should now be looking to see how we can append elements to a tuple, maybe sort it.

Tuples are Limited

Possibly we should now be looking to see how we can append elements to a tuple, maybe sort it.

We would be disappointed.

```
>>> k.sort ()
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'tuple' object has
no attribute 'sort'

>>> k.append ('Launcelot')
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'tuple' object has
no attribute 'append'
```

Tuples are Limited

Possibly we should now be looking to see how we can append elements to a tuple, maybe sort it.

We would be disappointed.

The Exceptions are telling us that the Tuple class has no such operations defined for it.

```
>>> k.sort ()
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'tuple' object has
no attribute 'sort'

>>> k.append ('Launcelot')
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'tuple' object has
no attribute 'append'
```

Tuples are Limited

Possibly we should now be looking to see how we can append elements to a tuple, maybe sort it.

We would be disappointed.

The Exceptions are telling us that the Tuple class has no such operations defined for it.

This is fine. A tuple is a collection where *order does not matter*, so sorting is irrelevant.

If we need to sort, we used a different data structure: a list.

```
>>> k.sort ()
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'tuple' object has
no attribute 'sort'

>>> k.append ('Launcelot')
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
AttributeError: 'tuple' object has
no attribute 'append'
```

Maths on Tuples

Actually, you can add to a tuple.

See that we have to add a *tuple*, not a string.

```
>>> k += ('Launcelot',)
>>> k
('Robin', 'Arthur', 'Galahad',
'Launcelot')
```

Maths on Tuples

Actually, you can add to a tuple.

See that we have to add a *tuple*, not a string.

We can also multiply tuples by integers.

```
>>> k += ('Launcelot',)
>>> k
('Robin', 'Arthur', 'Galahad',
'Launcelot')

>>> fearsome_cry = ('Ni!',)
>>> fearsome_cry *= 3
>>> fearsome_cry
('Ni!', 'Ni!', 'Ni!')
```

PyCharm Demo and Question Time

PyCharm Demo and Question Time

"What else can you do with lists?"

"How do you handle the error when the user enters a string and you want an integer?"

Step Back

We have now met strings and tuples.

They are both classes, but have different *attributes* and *methods*.

As far as is sensible, they work in the same way.

We should now understand how programs can store and manipulate data:

- ➢ Now we need to explore more about how to, for example, handle errors.

Jobs

By next week, you should:

- ➢ Have read up to the start of Unit 3 in the book.
- ➢ Practiced.
- ➢ Attempted the "Capstone Project" at the end of Unit 2 in the book.
- ➢ Practiced some more.
- ➢ Got answers to any questions you might have.