COMP10001 Foundations of Computing
String Manipulation and Conditionals

Semester 2, 2021
Chris Leckie, Marion Zalk and Farah Khan

THE UNIVERSITY OF MELBOURNE

— VERSION: 1485, DATE: MARCH 21, 2019 —

Lecture Agenda

- Last lecture — Grok Worksheets 1, 6
  - Variables and assignment
  - String basics
- This lecture — Grok Worksheets 3–4
  - String manipulation
  - Conditionals

Announcements

- Worksheets 0, 1 and 2 due end of this Friday ... don't need to do anything to "submit" your work; just get as many green diamonds as possible by then

What do We Know so Far?

Syntax

- Maths...
- `print()`, `len()`
- `int`, `float`, `str`
- `*`, `+` and `in` for strings
- Variables, assignment `=`

Semantics

- Maths expressions are resolved with BODMAS
- Types are important: overloading
- Assignment changes state

Class Exercise

- Given `num` containing an `int`, calculate the number of digits in it

Lecture Outline

1. Strings: Formatting
2. Strings: Indexing and Slicing
3. Conditionals

Strings and Formatting

- Often we want to insert variables into strings, optionally with some constraint on how they are formatted/presented
- We can do this in part through string concatenation (+), but it has its limitations:

```
>>> response = "yes"
>>> sentiment = 1/1
>>> print(response + ", " + response + ", " + \
... response + " ... I " + \
... str(100*sentiment) + "% agree")
yes, yes, yes ... I 100.0% agree
```

Strings and Formatting

- A cleaner, more powerful way is with **format strings** ("f-strings"), marked with an "`f`" prefix at the start of the string:

```
>>> response = "yes"
>>> sentiment = 1/1
>>> print(f"{response}, {response}, {response}" + \
... f" ... I {100 * sentiment:.0f}% agree")
yes, yes, yes ... I 100% agree
```

  - insert variables into strings with braces, possibly with some associated operators (e.g. `100 *`)
  - optionally add formatting specifiers with a colon ("`:`"), e.g. to stipulate the number of decimal places to use for a float (e.g. "`.0f`" = zero decimal places)

Lecture Outline

Sequences of Items

- One construct that pervades computing is a "sequence" (or "iterable" in Python-speak), i.e. the decomposition of an object into a well-defined ordering of items
  - text as sequences?
  - sounds as sequences?
  - images as sequences?
- Manipulation of objects tends to occur via "iteration" over iterables

String Manipulation

- As well as "assembling" strings via + and $*$, we are able to pull strings apart in the following ways:
  - "indexing" — return the single character at a particular location
  - "slicing" — extract a substring of arbitrary length
  - "splitting" — break up a string into components based on particular substrings

String Manipulation: Indexing

- Each character in a string can be accessed via "indexing" relative to its position from the left of the string (zero-offset) or the right of the string ([minus] one-offset):

I	t		w	a	s		a		d	a	r	k
0	1	2	3	4	5	6	7	8	9	10	11	12
−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

```
>>> story[-8]
's'
>>> story[5]
's'
```

String Manipulation: Slicing

- It is possible to "slice" a string by specifying a `START` and (non-inclusive) `END` `int` value:

```
>>> story[1:11]
't was a da'
```

  N.B. the sliced substring length = `END` − `START`

- By default, `START=0` and `END` is the length of the string:

```
>>> story[:-7]
'It was'
```

String Manipulation: Slicing

- It is also possible to specify slice "direction" (`1` or `-1`):

```
>>> story[-1:-7:-1]
'krad a'
```

Here, the first argument is still the START and the second is still the END, but the default values are `START=-1` and `END` = -(the length of the string + 1):

```
>>> s[-8::-1]
'saw tI'
>>> s[:-5:-1]
'krad'
```

Class Exercise

- Generate the "middle half" of a given string

Lecture Outline

In Search of the Truth ...

- Often, we want to check whether a particular value satisfies some condition:
  - does it have four legs?
  - is it over 18?
  - is it tall, with rabbit ears, a grey back, whiskers, a creme stomach with grey markings on it, and (at times) an umbrella?

Source(s): `http://fav.me/d4qp4si`

In Search of the Truth ...

- Often, we want to check whether a particular value satisfies some condition:
  - does it have four legs?
  - is it over 18?
  - is it tall, with rabbit ears, a grey back, whiskers, a creme stomach with grey markings on it, and (at times) an umbrella?

Source(s): `http://fav.me/d4qp4si`

In Search of the Truth ...

- For this, we require:
  - a way of describing whether the test is satisfied or not
  - a series of comparison operators
  - a series of logic operators for combining comparisons
  - a way of conditioning behaviour on the result of a given test

Capturing Truth: The `bool` Type

- We capture truth via the `bool` (short for "Boolean") type, which takes the two values: `True` and `False`
- As with other types, we can "convert" to a `bool` via the `bool()` function:

```
>>> bool(3)
True
>>> bool(0)
False
>>> bool("banana")
True
```

Every type has a unique value for which `bool()` evaluates to `False`

Evaluating Truth: Comparison

- We evaluate truth via the following Boolean comparison operators:

`==`	equality; NOT the same as =
`>`, `>=`	greater than (or equal to)
`<`, `<=`	less than (or equal to)
`!=`	not equal to
`in`	is an element of

```
>>> 2 == 3
False
>>> 'a' <= 'apple'
True
>>> 'bomp' in 'bomp, bomp, bomp'
True
```

Combining Truth

- We combine comparison operators with the following logic operators:
  - `and`, `or`, `not`:

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

not	True	False
	False	True

- NB: precedence: `not` > `and` > `or`

Lecture Summary

- What is a sequence/iterable?
- Strings: what are indexing, slicing and splitting?
- What is the `bool` type?
- What Boolean comparison operators are commonly used in Python?
- What logic operators are commonly used in Python? What is the operator precedence?
- What are `if` statements and code blocks?
- How can you cascade conditions in Python?