OSLOMET

Collections, Strings, and Files

Collating information

Nikolai Bjørnestøl Hansen
Roza Abolghasemi
OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Photo: Ronny Østnes / OsloMet

Follow along

- The easiest way to install Python on your computer is to install Anaconda from
  https://www.anaconda.com/products/distribution
- You can still follow along via an online interpreter if you want, like
  https://repl.it/languages/python

OSLOMET

Strings

A string is a collection of letters, surrounded by single or double quotes.

- We create strings by writing `s = "Hi, everybody!"` or `s = 'Hi, Doctor Nick!'`
- Or, if we want multiple lines of text, by writing

```
s = """This string will consist
of multiple lines
of text"""
```

- We can add two strings together, concatenating their values. So `"Hello, "+"world!" == "Hello, world!"` would return `True`.
- We don't need the +. Putting them next to each other also concatenates them.
- We can multiply a string and an integer, repeating the string. So `"Cookie!"*3 == "Cookie!Cookie!Cookie!"` would return `True`.

OSLOMET

Backslash (\)

The backslash character has special meaning inside of strings.

- `\n` means newline. The text continues on the next line.
- `\t` means tab. It inserts a horizontal tab into the text.
- `\"` and `\'` lets you use single and double quotes without ending the string. We can write `s = "He said \"I'll do it\""`.
- `\\` is used to insert a single backslash. Try printing `"The symbol \\n gives a newline."`
- `\N{name}` lets us print a unicode character with the given name. Try printing `"\N{cookie}"`.
- `\uxxxx` and `\Uxxxxxxxx` prints the character with `xxxx` or `xxxxxxxx` as its hex value. Try printing `"\U0001F36A"`.

The `r`-strings and the `f`-strings

- An `r`-string is a raw string. The special meaning of backslash is “turned off”.
- We create raw strings by prepending an `r` to the string. Try printing
  `r"Hello,\n and goodbye."`
- It is useful for representing Windows file paths (which use \). We will also use it for regular expressions, when we learn those.
- An `f`-string is a formatted string. It lets us insert code into a string.
- We create a formatted string by prepending an `f` to the string. We can then add code by surrounding it with curly brackets.
- Try printing `f"Seventeen times thirty-five equals {17*35}"`.
- Save your name in a variable called `name`, then try printing
  `f"My name is {name}"`.

OSLOMET

More about `f`-strings

- We can specify a format to the code in an `f`-string.
- This is done by using a colon.
- Try printing `f"Pi is equal to {3.14159:+07.2f}"`.
- It will print `"Pi is equal to +003.14"`.
- The `+` means “Print a plus for positive numbers.”
- The `0` means “Fill extra space after the sign with zeros.”
- The `7` means “The number will be 7 characters wide.”
- The `.2` means “Use 2 digits of precision.”
- The `f` means “Format it as a `float`.”
  - Other options here is `e`, scientific notation, `%`, as percent, and `g`, general form. The latter is the default.
  - For integers we can use `b`, `o`, and `x` to write it in binary, octal, or hexadecimal.
- Many other formatting options are also available.

OSLOMET

Other things to do with strings

- There is a myriad of other in-built string methods.
- Check out https://docs.python.org/3/library/stdtypes.html#string-methods for all the in-built methods.
- Some useful ones:
  - `str.count(sub)` counts how many occurences of `sub` is found within the string. So `"Hello from Hell".count("Hell")` would return `2`.
  - `str.find(sub)` tells you at what location in the string we can find `sub`. So `"Where's Waldo?".find("Waldo")` would return `8`.
  - `str.replace(old, new)` replaces each occurence of `old` with `new`. So `"I'm here!".replace("here", "gone")` would return `"I'm gone!"`.
  - `str.split(sep)` and `str.join(list)` splits a string into a list, or joins a list into a string. More on this after the next slides!

OSLOMET

Lists and tuples

- Lists and tuples lets us save multiple values in one object.
- We surround lists with square brackets and tuples with round brackets. Elements are separated by a comma.

```
a_list = ["This", "is", "a", "list"]
a_tuple = ("This", "is", "a", "tuple")
```

- The difference between them:
  - A tuple is constant. Once created, it cannot be changed.
  - Trying to change it will create an error.
  - Because it's constant, it will be slightly faster to fetch elements from it.
- We find the length of a list or tuple (or string!) with the `len` function.
- So `len(a_list)` returns `4`. And `len("How are ya?")` returns `11`.

OSLOMET

Fetching elements from lists, tuples, and strings

- We fetch elements from a list or tuple by using square brackets.
- Python starts numbering at zero!
- The code

```
a_list = ["This", "list", "has", "values"]
print(a_list[2])
```

would print `"has"`.

- If we ask for a negative position,we will count from the back.
- So `print(a_list[-3])` would print `"list"`
- Asking for an element outside of the list will give us an error.
- Python treats strings as a “tuple of characters”.
- With `s = "This is a list!"`, `s[0]` returns `"T"` and `s[-1]` returns `"!"`.

Slicing lists, tuples, and strings

- We fetch parts of a collection by slicing.
- If `a` is a list, tuple, or a string, we slice it by writing `a[start:stop]`.
- This returns a list, tuple, or string with all elements from position `start`, up to but not including position `stop`.
- `"How are you?"[2:5]` would return `"w a"`.
- One or both of these can be empty. That signifies it's most "extreme" value.
  - `"012345"[3:]` would return `"345"`.
  - `"012345"[:3]` would return `"012"`.
  - `"012345"[:]` would return `"012345"`.
- We can also add a "step". `"012345"[1:5:2]` would return `"13"`.
- The step can be negative! `"012345"[5:0:-1]` would return `"54321"`.
- Reversing a list, tuple, or string can therefore be done with `a[::-1]`.

OSLOMET

Splitting and joining

- Given a string, we can turn it into a list with the `split`-method. It splits the string based on a separator.
- The code `"Bananas".split("a")` returns `["B","n","n","s"]`.
- If given no separator, the method splits based on whitespace.
- The code `"This is a list".split()` returns `["This", "is", "a", "list"]`.
- The code `"This\nis \ta list ".split()` gives the same result.
- Given a list of strings, we can join them into a single string with the `join`-method.
- The code `"a".join(["B","n","n","s"])` returns `"Banana"`.
- The code `"".join(["Bob","ert"])` returns `"Bobert"`.

Dictionaries

- Lists and tuples lets you fetch items by position. Dictionaries instead lets you fetch them by name.
- The name could be anything, but is usually a string.
- We create a new dictionary like this:

```
d = {"pi": 3.14, "tau": 6.28, 2.72: "e"}
```

- The “name” is called the key, and the item is called the value
- We fetch the value by writing `d["pi"]` or `d[2.72]`.
- We can also get values by writing `d.get("tau")`.
- Using `get` would return `None` if the key doesn’t exist. Using square brackets would instead throw an error.

OSLOMET

More on dictionaries

- We can add to or change a dictionary by assigning values to the keys.
- Try the following code:

```
car = {"brand": "Ford", "model": "T", "year": 1908}
car["color"] = "black"
car["model"] = "Tin Lizzie"
print(car)
```

- We remove a element from a dictionary by popping it, `car.pop("year")`. This also returns the value.
- We can also update values with the `update`-method. This method takes a dictionary as input.
- The two middle lines in the code above could have been written as `car.update({"color": "black", "model": "Tin Lizzie"})`.

OSLOMET

The `in` keyword

- To se if something is in a collection, we have the `in`-keyword.
- It checks if an element is in a list or tuple, if a key is in a dictionary, or if a substring is part of a string.
  - `3 in [1, 2, 3, 4, 5]` returns `True`.
  - `"read" in "Tread lightly"` returns `True`.
  - `"color" in {"color": "black", "model": "T"}` returns `True`.
  - `"T" in {"color": "black", "model": "T"}` returns `False`.
- We use this with `if`-tests and `while`-loops.

```
car = {"brand": "Ford", "model": "T", "year": 1908}
if "color" in car:
    print(f"The color of my car is {car["color"]}")
else:
    print("My car seems to be colorless")
```

OSLOMET

The for-loop

- A `while`-loop checks a condition to see if we want to continue looping, but often we want to loop through a collection.
- For this we have the `for`-loop.
- Try this code:

```
l = ["A", "list", "of", 3.14, "different", ["hey", "a", "list"], "things"]
for i in l:
    print(i)
print("And we're done!")
```

- It loops through the collection, in each loop the value `i` represents the current collection-value.
- If we know that we want to loop, say, 100 times, we often use the `for i in range(100):`-construction. Here `range(100)` is essentially a list `[0, 1, ..., 99]` of 100 elements.

OSLOMET

Breaking loops

- If we are in the middle of a loop, we have access to two extra keywords, `break` and `continue`.
- The keyword `break` stops the loop completely.
- Try the code

```
string = "Papaya Fruit"
for letter in string:
    if letter == "r":
        break
    print(letter)
```

- It should print each letter up to but not including the `r` in `Fruit`.
- If we have a loop inside of a loop, the `break` keyword will only stop the current loop. The outer loop would keep running.

OSLOMET

Continuing loops

- The second keyword, `continue`, skips to the next step in the loop.
- Try the code

```
string = "Papaya Fruit"
for letter in string:
    if letter in "aeiou":
        continue
    print(letter)
```

- This should print each consonant in the string.
- Every time the loop finds a vowel, it'll continue on to the next run of the loop, and therefore skipping the print statement.
- Breaking and continueing can be done in both `for`-loops and `while`-loops.

OSLOMET

Opening and closing files

- To open a file, we use the `open` method. We write `file = open(filename, mode)`.
- The mode can be `"r"` for read, `"w"` for write, `"a"` for append, or `x` for create.
- If you open a non-existent file in create, append, or write-mode, a new file will be created. If you open a existent file in create mode, an error will be thrown.
- If you don't add a mode, the file will open in read mode.
- When you're done with a file, you need to remember to close the file again with `file.close()`. If you don't close it, some changes may not be saved.
- The `with` keyword can be used to automatically close a file.
- The code

```
with open("notes.txt", "w") as file:
    file.write("Hey i've made some notes")
print("The file is closed now")
```

closes the file for you.

Reading from files

- Once a file is opened in read mode, we can read the entire file with `file.read()`.
- This gives us a single string with all the text in the file. Often we would rather cut the file into smaller, more manageable chunks.
- The command `file.readline()` reads a single line from the file. The next time you call `file.readline()`, you'll get the second line, and so on.
- Python thinks of files as collections of lines. We can use this to loop through the whole file with a `for`-loop, like this:

```
with open("notes.txt") as file:
    for line in file:
        print(line, end="")
print()
print("The whole file has now been printed")
```

OSLOMET

Writing to files

- Writing to files is done via the `write` method.
- The code `file.write("It was the best of times, it was the worst of times")` writes that text string to the file.
- No newlines are added, so the next `file.write` will continue exactly where the last left off.
- If the file is in append-mode, it'll be added onto the end of the file.
- If the file is in write-mode, it'll overwrite the file.
- If the file is in create-mode, the file will be a new file.
- There is a last mode, `"r+"`. It lets you both read and write to a file. The writing continues from where you've read to.
- If you haven't read the whole file, this will probably overwrite something you'd like to keep.

OSLOMET