IT@JCU

CP1401 Module 9
Strings & Files

JAMES COOK
UNIVERSITY
AUSTRALIA

Learning outcomes - you will be able to:

- Process (input/data) strings
  - Often, data is provided not as separate values in neat variables, but as combined strings (or text files) that need to be processed. We can process strings to extract specific data from general text.

- Read and write data from and to text files
  - Without file I/O (input/output), we can't store data persistently – our programs will 'forget' everything when they finish.
  - With files, we can access (read) and store (write) large amounts of data very quickly.

Do this now

- Write an algorithm to get and display only the year from a date of birth stored as a string like "13/07/1995"

- Write an algorithm for a function to determine if a string date of birth (like above) matches the pattern "dd/mm/yyyy"
  - "22/01/2015" does match
  - "12/03/98" and "12/3/1998" do not match
  - don't worry about checking actual dates, so "99/77/9876" matches, but "dd/1g/-$#3" does not

In the Lists lecture, we saw that strings, like lists, are sequences

Strings are sequences too

- Much of what works with lists also works with strings (but not modifying, because strings are **immutable**)

```
string = "Hello"
print(string[0])  # 'H'

for character in string:
    print(character.upper(), end="-")
print(len(string))
# Prints H-E-L-L-O-5
```

Each character in a string has an index

(each item in any sequence)

```
text = "ABC"
```

Index	Element
0	"A"
1	"B"
2	"C"

- Index of the first element (character) in the string is 0,
  second element is 1, and
  nth element is n-1
- Negative indexes identify positions relative to the end of the list (not available in many other languages)
  - -1 identifies the last element, -2 identifies the second-last element, etc.

Index	Element
-3	"A"
-2	"B"
-1	"C"

Use indexing to access characters in a sequence

```
text = "ABC"
print(text[2])  # Prints "C"
```

- Use the index in square brackets to access individual elements

What is the output of this code?

```
text = "Programming is fun"
print(text[1], text[-1], text[-3])
```

Beware of invalid indexes

- An IndexError exception is raised if an invalid index is used

```
text = "one"  # only three elements
print(text[3])  # accessing non-existent element
```

```
IndexError: string index out of range
```

Use slicing to access slices of a sequence (string)

- Like indexing, slicing works for lists, tuples, strings - any sequence
- A slice is a span of items taken from a sequence
  - Known as a substring for string slices
- Slicing format: `sequence[start:end]`
  - Expression will return a sequence containing a copy of the elements from start up to, but not including, end (very similar to `range`)
  - If start is not specified, 0 is used for start index
  - If end is not specified, len(string) is used for end index
- Slicing expressions can include a 3rd step value (also like range) and negative indexes relative to end of string

What are these slices (substrings)?

```
text = "Programming is fun"  # 18 characters
```

	Expression	Value
1	`text[0:4]`	
2	`text[4:7]`	
3	`text[0:-4]`	
4	`text[0:-4:2]`	
5	`text[0:len(text)]`	

String methods, functions, operators

Strings (and most other types) have **methods**

- We have seen methods before:

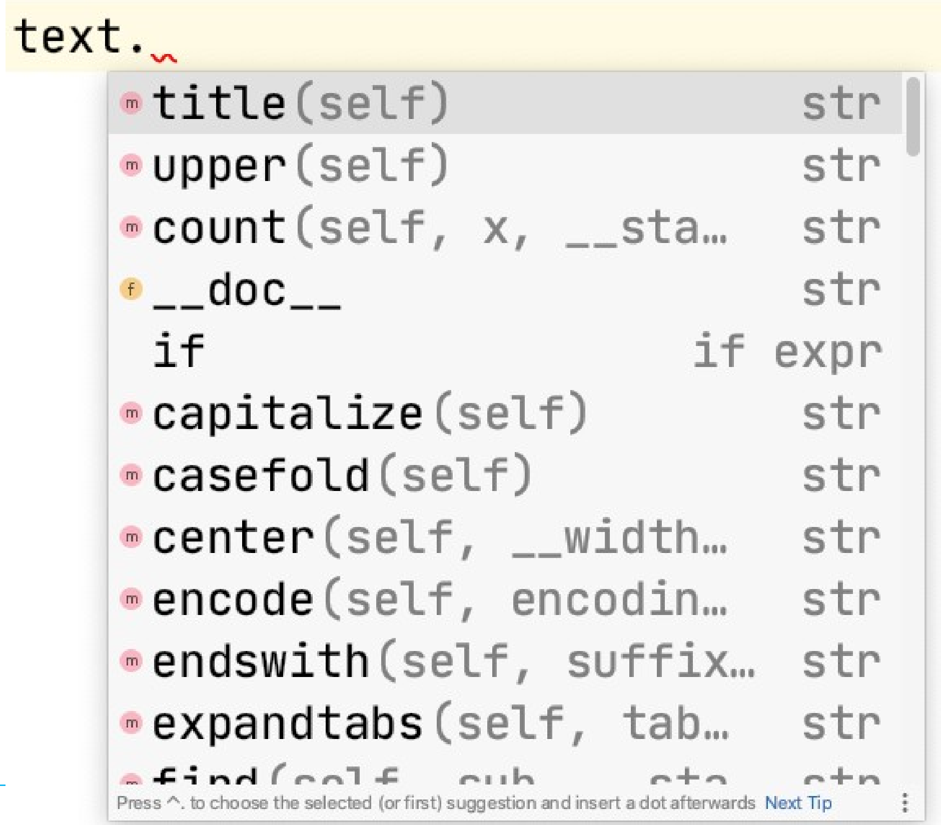
  ```
  choice = input("Choice: ").lower()
  ```

  - Remember how the dot separates the **object** (left) from the **method** (right)

Some useful string methods include

- **upper**(), .**lower**(), .**title**()
  get string in a different case

- Press . and view the popup to see more
  and try them out! (in the console)



```
text.
title(self)                    str
upper(self)                    str
count(self, x, __sta…          str
__doc__                        str
if                         if expr
capitalize(self)               str
casefold(self)                 str
center(self, __width…          str
encode(self, encodin…          str
endswith(self, suffix…         str
expandtabs(self, tab…          str
find(self, sub, sta…           str
```

Press ^. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

Python has methods to search for substrings

- **endswith**(substring), **startswith**(substring): check if the string ends or starts with substring
- **find**(substring): searches for substring within the string
  - Returns lowest index of the substring,
    or if the substring is not contained in the string, returns -1

```
if subject_code.startswith("CP"):
    print("That's an IT subject :)")
```

Do this now

- Rewrite this code to work with **slicing** instead of startswith()

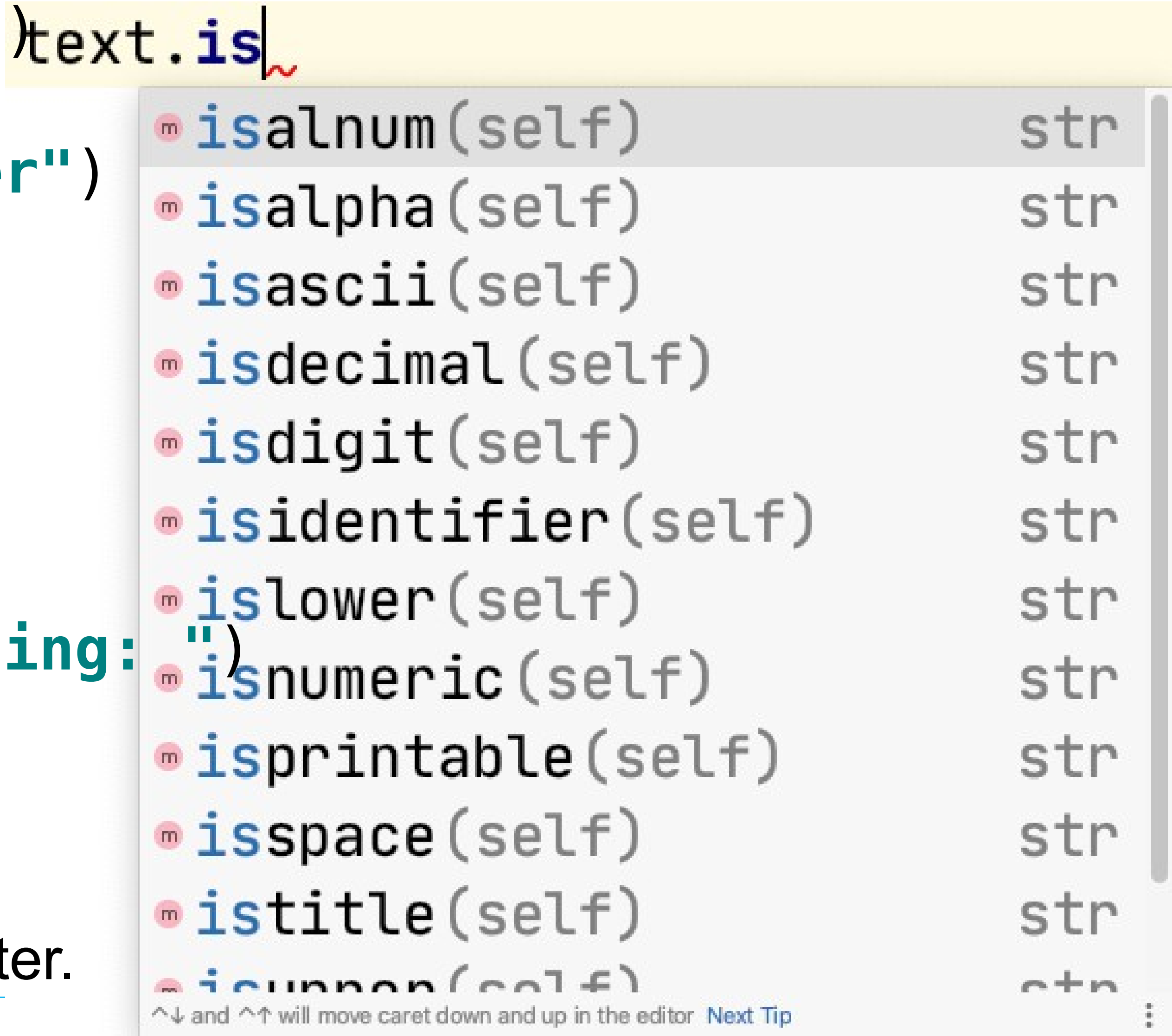
```
if subject_code.startswith("CP"):
    print("That's an IT subject :)")
```

Test if a string is… something with these methods

```
phone_number = input("Phone: ")
if not phone_number.isdigit():
    print("Invalid phone number")

name = input("Name: ")
if not name.isalpha():
    print("Alphabetical only")

expression = input("Say something: ")
if expression.isupper():
    print("No need to shout")
```



```
text.is
```

These are often/usually used per character.

Use "in" to determine if a substring is in a string

- The **in** and **not in** operators can be used for searching a collection

```
if "CP" in subject_code:
    print("You are cool")

if needle not in haystack:
    print(value, "is not found")

"ab" in "able"   # True
"AB" in "Able"   # False
```

Don't use "in" to check equality

- The **in** and **not in** operators are for looking "in", not comparing to

```
if required_password in your_password:
    print("Access granted!")
```

Oh no!

```
"secret" in "not_secret"  # True
"secret" == "not_secret"  # False
```

String + String = String

- **+** is the "concatenation" operator
- **+=** adds to the end of a string
    - It actually creates a new string set to the new value

```
phone_number = "07" + phone_number
expression += "!"
```

- How could we make sure a sentence starts with a capital and ends with a full stop… and change it if needed?

"Escape characters" allow you to create special strings

- How could we print the following in one statement?

```
JCU Douglas Campus
Townsville
    QLD
```

- Newline characters are `\n` - `"First Line\nSecond Line"`
- Tabs are `\t`
- Printing " inside double quotes… use `\"`

```
print("JCU Douglas Campus\nTownsville\n\tQLD")
```

Python has built-in functions that work with strings

- **len**, as expected
- **min**, **max** return the alphabetical lowest or highest characters
- **sum**? Doesn't make sense

- What else can you find?

We should now be equipped to solve these:

- Get and display only the year from a date of birth stored as a string like "13/07/1995"
- Write a function to determine if a string date of birth (like above) matches the pattern "dd/mm/yyyy"
  - "22/01/2015" does match
  - "12/03/98" and "12/3/1998" do not match
  - don't worry about checking actual dates, so "99/77/9876" matches, but "dd/1g/-$#3" does not

Do this now

Suppose you have a phrase from the 'name game' like:

```
text = "Hi my name is Jim and I like jumping"
```

or

```
text = "I'm Betty and I like bowling"
```

We want to find just what the person likes ("jumping" or "bowling")…

Write a program to process text like this and print just the liked thing.

Hint: the find method should be useful.

Files

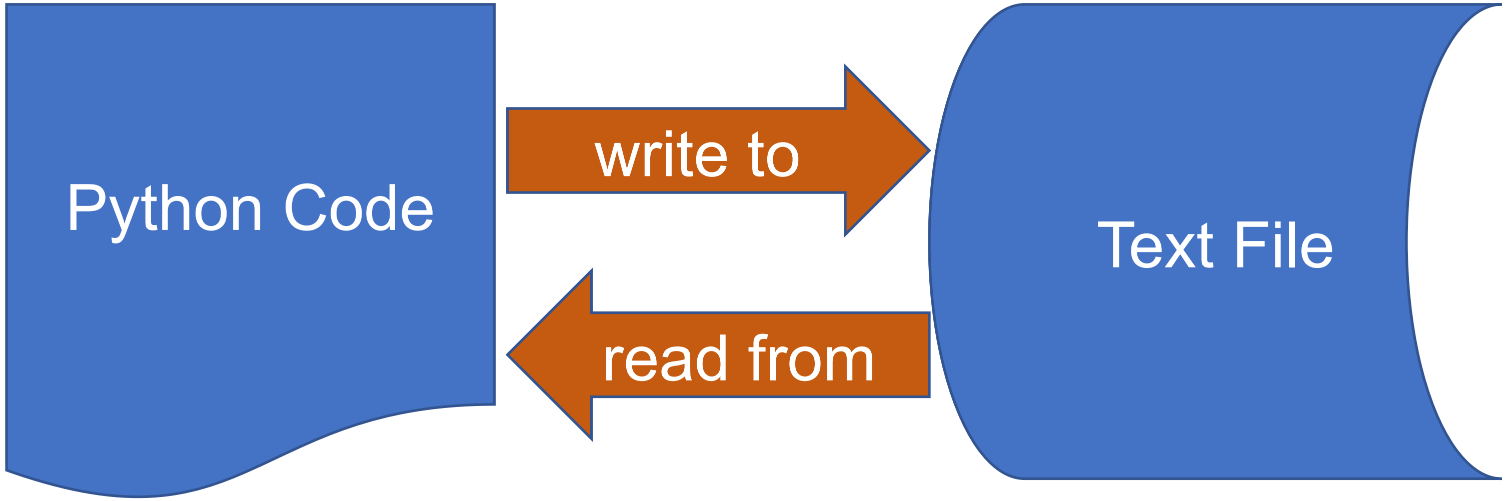
Do this now

- Write a program to get strings from the user until they enter an empty string.
  When they have finished, display only the ones that started with a capital letter.

□ *Hmmm…*
How would we store those strings for later… not just for now?

Files allow us to retain data outside programs

- Until now, our program's data only existed in memory - temporary
- We want to save our data by **writing** to an output file
- We want to retrieve our data by **reading** from an input file
- These can be the same file. We will only work with text files.



Interacting with files always follows these 3 steps

1. Open the file
2. Process the file (read or write)
3. Close the file

Open a file in Python with the **open** function

- The open function creates a **file object** and associates it with a file on the disk

```
file_object = open(filename, mode)
```

- The mode specifies how the file will be opened.
  There are other modes, but for now we are only interested in:
  - 'r' = read (the default)
  - 'w' = write
  - 'a' = append (write to the end)

```
in_file = open(FILENAME, "r")  # Open for reading
out_file = open("data.txt", "w")  # Open for
writing
```

Always **close** open files

- When you're finished reading or writing data to a file, you should always close it:

  ```
  file_object.close()
  ```

- Failing to close a file could result in a loss of data or the file being inaccessible until the program closes
  - You can not read from, or write to, a closed file

```
out_file.close()
in_file.close()
```

Read the contents of a file with the **read** method

- The read method reads the entire file contents into one string
  - Only works if file has been opened for reading ('r')

```
FILENAME = "testfile.txt"
in_file = open(FILENAME)
text = in_file.read()
in_file.close()
print(text)
```

Remember our guessing game?
(and the random secret version)

```
SECRET = 6
guess = int(input("? "))      ← "priming read"
while guess != SECRET:        ← meaningful (readable) condition
    print("Guess again!")
    guess = int(input("? "))  ← same as priming read
print("You got it!")
```

- **This is THE standard while loop format we want you to learn.**
- Note that the line before the loop header (condition) is the same as the last line of the body

Now let's read the secret number from a file

```
in_file = open("secret.txt")
secret = int(in_file.read())
in_file.close()

guess = int(input("? "))
while guess != secret:
    print("Guess again!")
    guess = int(input("? "))
print("You got it!")
```

Here we assume that the contents of the file is a single integer.

Use a for loop for reading each line in a file

- Python's for loop iterates through each line in a file, one at a time
  - "each" line = definite iteration
  - This is fantastic 🏝
- Useful when you want to do the *same thing* with each line
- Not useful when you want to do *different things* with different lines

```python
in_file = open("letter.txt")
for line in in_file:
    print(line)
in_file.close()
```

`line` is a good variable name for a… line!

Strip whitespace with the **strip** string method

- The format of an output file depends on how you write to it (just like printing) – the spaces and newline characters matter
- Same for reading – you need to know the format to read it
- You often need to remove `'\n'` from strings read from a file
  - `strip` method strips whitespace from both start and end of a string

```python
in_file = open("letter.txt")
for line in in_file:
    print(line.strip())
in_file.close()
```

Do this now

- Write a program to read the file "letter.txt" and print ONLY the lines that start with a capital letter
- Remember:
  - strings can be sliced and indexed
  - strings have useful methods, including ones for testing

When you open a file with 'w' mode...

- if the file already exists:
  - any existing data in the file is overwritten
- if the file does not exist:
  - Python creates it (in the current project/code folder)
- if you want to add data to the end of an existing file (e.g., a log file), you can use the 'a' mode for "append".

You can write to a file with the print function

- Once you have created a file object, opened for writing, you can use print to write to it by adding the keyword argument like `file=file_object`

```
name = input("Name: ")
out_file = open("name.txt", "w")
print(name, file=out_file)
out_file.close()
```

When using functions with files...

- Remember the Single Responsibility Principle (SRP):
  - The function should do "one" job
  - The function should do the whole job
  - Functions should be reusable where possible

- **Don't**: write functions that only do part of one job by opening files outside and passing an open file object into a function
- **Do**: write functions that do the whole job (and are reusable) by passing in a file name and having the function open, process, close.

Do the whole of one job in a reusable way (SRP)

Poor design

```
def main():
    filename = "file.txt"
    out_file = open(filename, 
'w')
    process(out_file)
    out_file.close()

def process(file_object):
    for i in range(99):
        print("problem", 
file=file_object)
```

Good design

```
def main():
    filename = "file.txt"
    process(filename)


def process(filename):
    out_file = open(filename, 
'w')
    for i in range(99):
        print("problem", 
file=file_object)
    out_file.close()
```

Now do these next steps

- Practise writing algorithms and programs that use string processing and files
- "Play" in the Python console with string processing, practising with the things you've learned and experimenting with other things you can find
    - E.g., can you add two strings together? What about a string * a number? How could you create a 2nd string that was the same as a first but each character's case was inverted ("Hello!" becomes "hELLO!")? Or…?
- Look forward to Programming 2 where you will learn and use even more ways to process strings and files.