CMPUT 274

Modules, File I/O, Dictionaries, and other bits

Topics Covered:

- map function
- import modules
- reading/writing to files
- dictionaries

MORNING PROBLEMS: `map()`

- Applies given function to every item of an iterable object

```
map(function, iterable, ...)
```

- Returns a map object (an iterator)
- To use map object:

1. Can unpack elements of map object and assign to individual variables

```
# x1, y1 = map(int, input().split())
line1 = input()

line1_list = line1.split()

x1, y1 = map(int, line1_list)
```

line1 is a string: e.g. `"-4 -2"`

line1_list is a list of strings: e.g. `["-4", "-2"]`

line1_list[0] converted to integer and assigned to x1 `(=-4)`
line1_list[1] converted to integer and assigned to y1 `(=-2)`

MORNING PROBLEMS: `map()`

- To use map object (con't):

  2. Can convert map object to a list

```
# coords = list(map(int, input().split()))
line1 = input()

line1_list = line1.split()

coords = list(map(int, line1_list)
```

line1 is a string: e.g. `"-4 -2"`

line1_list is a list of strings: e.g. `["-4", "-2"]`

line1_list[0] converted to integer;
line1_list[1] converted to integer;
Map object converted to list e.g. `[-4, -2]`

Modules

Modules

- **Module**: file that contains Python function definitions
  - Another Python program will import module and call functions
  - Allows functions to be reused
  - Organization: related functions grouped in one file
- Module can also contain objects that can be accessed from other files (generally constants)
- Rules for module names:
  - Filename should end in .py
  - Cannot be the same as a Python keyword

Modules

- To use contents of module in another Python file, use `import` statement

```
# use any function in module
import module_name
module_name.function1()

# import specific functions in module
from module_name import function1, function2
function1()
```

Use dot notation when calling function

Call function without reference to module

__name__ == "__main__"

- Can include additional code outside of function definitions in module file
- Good idea to use

```
if __name__ == "__main__":
    # code to run when this is main program
```

- When module file is run, it is the main program
  → all code under `if __name__ == "__main__"` will run
- When module file is imported, it is NOT the main program
  → code under `if __name__ == "__main__"` will NOT run

Example

- rectangle.py

```
def area(width, length):
    return width*length

if __name__ == "__main__":
    print("Inside rectangle.py")
    print(area(3,5))
```

```
Inside rectangle.py
15
```

- soccer_field.py

```
import rectangle

if __name__ == "__main__":
    print("Inside soccer_field.py")
    print(rectangle.area(70, 100))
```

```
Inside soccer_field.py
7000
```

Standard library modules

- Python has a library of pre-written functions stored in modules that are installed with Python
- These library functions perform tasks that programmers commonly need
- Examples:
  - time
  - sys
  - os
  - math
  - random
  - pickle
  - etc.

`math` Module

- https://docs.python.org/3/library/math.html
- Rounding functions
  - e.g.: math.ceil(x), math.floor(x), math.trunc(x)
- Power and logarithmic functions
  - e.g.: math.exp(x), math.log(x), math.log10(x), math.sqrt(x)
- Trigonometric functions
  - e.g.: math.cos(x), math.sin(x), math.tan(x)
  - e.g.: math.acos(x), math.asin(x), math.atan(x)
  - e.g.: math.degrees(x), math.radians(x)
- Constants
  - e.g.: math.pi, math.e

Example: `math` Module

```
import math

radius = int(input("Enter circle's radius: "))
circle_area = math.pi * math.pow(radius,2)
print('The area of that circle is', circle_area)
```

```
Enter circle's radius: 2
The area of that circle is 12.566370614359172
```

`random` Module

- Can use to generate pseudo-random numbers
- See descriptions in Python Intro Lab (Lab 3), Weekly Exercise #2
- Refer to official documentation: https://docs.python.org/3/library/random.html

File I/O

Files for Input/Output

- So far, we have received input from user via the keyboard → `input()`
- But some problems require a lot of data, or the same data to be reused
  - Manually entering data can be tedious/unrealistic for user
  - Instead, save data to a file
  - Allows program to retain data between executions
- In general, 2 types of files:
  - Binary
  - Text (human readable)

Using Files

- Three main steps to using files:

  1. Open a connection to a file.
     → Create file object

  2. Read data from file or write data to file.

  3. Close connection to file. ⇦ *** Don't forget! ***

Open File

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

- Modes for opening files:
  - Read only (default) ➔ "r"
  - Read and write ➔ "r+"
  - Write only ➔ "w"
  - Append to the end of file ➔ "a"
  - Append a "b" to above modes for binary file, "t" for text (default)

```
# opens file to read from
fin = open("studentData.txt", "r")
```

- Provide absolute or relative path in filename

`os.path`: Check if File Exists

- Before trying to open a file, may want to check if the file exists

- Use `os.path` module:

  `os.path.isfile(fname)`

  → Returns True if `fname` exists

```
import os.path

fname = input("Enter a filename: ")
while not os.path.isfile(fname):
   print("File does not exist")
   fname = input("Enter a filename: ")

fin = open(fname, 'r')
```

Methods to Read from File

- Three methods to read from a file:

1. *file_object_name*.**read**(*size*)
   - Reads contents of file up to `size` characters (text file)
   - If size is not specified, will read to end of file
   - Contents returned as a single string, including any \n

2. *file_object_name*.**readline**()
   - Reads a single line
   - Line is returned as a string, including \n if present

3. *file_object_name*.**readlines**()
   - Reads all lines in file
   - Lines returned as list of strings, including \n if present

Examples: Reading from File

```
# Example 1
infile = open('names.txt','r')
for line in infile:
    line = line.strip('\n')
    print(line)
infile.close()
```

Example #1 views file as a list

```
# Example 2
infile = open('names.txt','r')
alist = infile.read().splitlines()
for line in alist:
    print(line)
infile.close()
```

Example# 2 reads the file into a list

Both examples produce identical output

Writing to File

file_object_name.**write**(*string*)

- Used to write data to a file, or append data to a file
  → depends on mode file was opened in

- Argument must be a single string
  → use `str()` to convert

Close File

- **Always close any file you open!**
  - write: closing file flushes buffer
  - read: hogs resources if you don't close

- Get in the habit of writing
  `file_object_name.close()` after opening a file
  - then fill in lines of program in between

- OR use context manager to automatically close file when finished using ← best practice

```
with open("studentData.txt", "r") as fin:
    my_data = fin.read()
```

Dictionaries

Built-in type: Dictionary

- **Dictionaries** are collections of associated pairs of items

```
capitals = {
     "AB":"Edmonton",
     "BC":"Victoria",
     "ON":"Toronto"
  }
```

  - Dictionaries are mutable
  - Elements in a dictionary do <u>not</u> have an order

  - A pair consists of a key and a value {key: value}
  - Key must be unique and immutable
  - Value can be non-unique and immutable or mutable

Built-in type: Dictionary

- Values are accessed via their keys: `capitals["AB"]`
- New pairs can be added: `capitals["QC"]="Montreal"`
- Existing values can be changed: `capitals["QC"]="Quebec"`
- Existing pairs can be deleted: `del capitals["BC"]`

AB:Edmonton	QC:Quebec	ON:Toronto

- `list(`*`dict_name`*`)` returns a list of keys of dictionary
- *`dict_name`*`.keys()` returns iterable keys of dictionary
- *`dict_name`*`.values()` returns iterable values of dictionary
- *`dict_name`*`.items()` returns iterable pairs (key, value) of dictionary
- `in` returns True or False depending on whether the key exists