Chapter 8: Additional Techniques and Structures

CSCI-UA 0002 – Introduction to Computer Programming
Mr. Joel Kemp
Agenda

• File I/O Write and Pickles
• in Lists
• List Comprehensions
• map()
• Lambdas
• Slicing
• String Iteration
FILE I/O – WRITING & PICKLING
• How do you write data to a text file?
  – Open the file with write permissions (mode)
    ```python
    file = open("data.txt", "w")
    ```
    • File gets created if it doesn’t exist – deleted if it does exist.
  – Write a **string** to the file using the `write()` function.
    ```python
    file.write("This is my string!")
    ```
    • If you want newlines, you have to put them manually `\n`
  – Close the file, as usual.
    ```python
    file.close()
    ```

• File permissions:
  – “r” : read only. The default if you don’t supply any.
  – “w” : overwrites what’s in the file for each run of the program.
    • i.e., existing files with the same name will be erased.
  – “a” : appends data to the end of the file.

Refer to: `write.py` and `output.txt`
Pickling

- Pickles are a means of file storage!
  - They allow you to save complex structures as strings.
    - You don’t have to worry about *serialization*.
      - **Serialization**: the process of converting a data structure or object state into a format that can be stored.
    - You also don’t have to worry about *deserialization*.
      - **Deserialization**: reading a complex structure from the file.

- Import the library, `pickle`.
- File extension: `.p`
- How do we use it?
  - Open a file in **binary** write mode: “*wb*”
    ```python
    file = open(“output.p”, “wb”)
    ```
  - Write the container to file using `pickle.dump()`
    ```python
    pickle.dump(some_container, file)
    ```
  - Load the container from file using `pickle.load()`
    ```python
    some_container = pickle.load(file)
    ```
- **Note**: file needs to be a new file handle opened with “*rb*” permissions!

Refer to:
- pickles.py and output.p
Chapter 7 from Visual Quickstart Guide

LISTS
in Lists

• Given a list: \( \text{data} = [1, 2, 3, 4] \), write code to determine whether or not the value 3 exists in the list.

  – Print \texttt{True} if the value is in the list, otherwise print \texttt{False}.

Solution:

\[
\text{num\_found} = \text{false}
\]

\[
\text{for } \text{num} \text{ in } \text{data}:
\]

\[
\text{if}(\text{num} == 3):
\]

\[
\text{num\_found} = \text{True}
\]

print (\text{num\_found})

Alternative:

\[
\text{num\_found} = 3 \text{ in } \text{data}
\]

print (\text{num\_found})

• You can also use the in operator as part of expressions in branching and looping.
Exercises

1. Given a dictionary, `phone_book`, of phone numbers indexed by names (assume unique names), write a function that does the following:

   ```python
   get_phone_number(phone_book, name)
   ```
   
   – If the name exists in the dictionary, return the phone number.
   
   – Otherwise, return `-1`.

   – **Note:** you can’t just say `phone_book[name]`.
     
     • If the value of name isn’t a key, then you’ll get a `KeyError` exception.
     
     • You need to check if the name is a key first before trying to get the value.

Refer to: `inexercises.py`
List Comprehension

• What is it?
  – A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.
    • Creates a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

• How do you use it?
  my_list = [exp for ... in ...]
  my_list = [exp for ... in ... if (...)]
  my_list = [func(exp) for ... in ... if (...)]

• Uhhhh, why?
  – It’s more concise and readable!
Comprehensions

• Given a list: \( \text{data} = [1, 2, 3, 4] \), write a program to create a new list \( \text{evens} \) that consists of the even numbers within \( \text{data} \).

**Solution:**

```python
evens = []
for num in data:
    if num % 2 == 0:
        evens.append(num)
```

**Alternative: List Comprehension**

```python
evens = [num for num in data if num % 2 == 0]
```

• List comprehensions allow you to construct lists in an easier and more readable fashion.
More Comprehensions

• Generate a list consisting of the squares of numbers from 1 to 10.
  
  Solution:
  squares = []
  for num in range(1, 11):
      squares.append(num ** 2)

  Alternative: List Comprehension
  squares = [num ** 2 for num in range(1, 11)]

• Generate a list of cubed primes from 1 to 100!
  
  Solution:
  cps = []
  for num in range(1, 101):
      if prime(num):
          cps.append(num ** 3)

  Alternative: List Comprehension
  cps = [num ** 3 for num in range(1, 101) if prime(num)]
Exercises

1. Rewrite the following comprehension into the non-comprehension form:
   
   \[
   S = [2 \times x \text{ for } x \text{ in range}(101) \text{ if } x \times 2 > 3]
   \]

2. Write a function, `pos_div_n()`, that takes in a list of numbers, `data`, and an integer `N`.
   - This function should return a list of numbers from `data` that are positive and divisible by `N`.
   - You can still have boolean operators in the comprehension’s `if` statement.

Refer to:
`compexercises.py`
map()

- The map() function lets you apply a function to every element in a list.

\[ \text{almost\_list} = \text{map}(\text{function, list}) \]

- Returns a list of results of applying a function on each element.
  
  - Actually, it returns a map object that you must convert to a list...
  
  - You pass in a function identifier, not a function call!

- Given a list, \( \text{data} = [1, 2, 3, 4] \) determine whether or not each element is prime.

\[ \text{is\_primes} = \text{map}(	ext{prime, data}) \]

\# \text{is\_primes} = [False, True, False, True]

- Note that we’re passing \text{prime} and not \text{prime()}

- What if the function that I’m passing in expects more than one list input?

  - You can add it as another input to map()

\[ \text{bars} = \text{map}(\text{foo, data1, data2}) \]

Refer to: maps.py
More $\text{map()}$

- Sometimes, it may seem like you can just use list comprehensions instead of $\text{map()}$:  
  
  ```python
  # Assume square() exists and data is a list
  r = list(map(square, data))
  
  r = [square(num) for num in data]
  ```

- How would a comprehension achieve this?

- In fact:
  - The two methods, $\text{map()}$ and comprehensions, are interchangeable.
  - $\text{map()}$ exists in other programming languages!
    - List comprehensions typically don’t exist.
Lambda

• It kind of sucks that you need a dedicated function to use within map...

```python
r = list(map(square, data))
```
– This means that `square` has to be defined as a separate function in the program...
  • Even if you’re only planning on using it once.

• Can’t we avoid that clutter and just temporarily define a function so that `map()` is happy?
  – But wait... why don’t we just stop using `map()`?
    • You lose that expressive power and have to write more code! 😞
  – Temporary (anonymous) functions are known as `lambdas`!

• Syntax:

```python
# Same functionality as square()
lambda num: num * num
```
– `lambda` keyword followed by parameter(s)
– Colon to indicate a block
– Function implementation follows the colon
More Lambda

\[ \text{\texttt{lambda num: num } \ast \text{num}} \]

• Let’s use this anonymous function in \texttt{map()}:  
  \[ r = \text{list}(\text{map(\texttt{lambda num: num } \ast \text{num}, \text{data}))} \]
• What if I want to take in more than one parameter?  
  \[ \text{\texttt{lambda n1, n2, n3: n1 + n2 + n3}} \]
• But what if I want to do something more complex than squaring?  
  – If your function performs more than a single step, you need to define the function separately. 😞  
  – You can call a function within the body of the lambda:  
    \[ \text{\texttt{lambda elem: square(elem)}} \]
• Lambdas are great for really short functions.  
  – You’ll lose the ability to reuse the lambda though 😞  
  – Dedicated functions were great because you can call them whenever you need them.
• Your parameters can be lists, dictionaries, strings, or anything else that a regular function could accept.
Slicing

- Slicing is a convenient way to request multiple elements within a list.
  - Returns a list with the elements of the sliced positions.

  How do you use it?

  a[start:end]  # items start through end-1
  a[start:]     # items start through the rest of the array
  a[:end]       # items from the beginning through end-1
  a[:]          # a copy of the whole array

- You can also have negative indices!

  a[-1]        # last item in the array
  a[-2:]       # last two items in the array
  a[:2]        # everything except the last two items
String Iteration

• You can loop through a string!
  – Gives you character-level access to the data.
• A string really is a list of characters.
  – i.e., we’re looping through a normal list.
• Example:
  ```python
  my_string = "Joel"  # ["J", "o", "e", "l"]
  for c in my_string:
      print(c)
  ```
• Alternative:
  ```python
  for i in range(len(my_string)):
      print(my_string[i])
  ```
• When is this helpful?
  – Searching for a particular character/delimiter.
Exercises

1. Write a function `front_back` that takes in a string and returns a new string where the first and last characters have been exchanged!

   `front_back('code')` → 'eodc'
   `front_back('a')` → 'a'
   `front_back('ab')` → 'ba'

Refer to:
stringExercises.py