CS134 Lecture 7: Lists, Ranges and Loops

Announcements & Logistics

- **Lab 3** was released Friday
  - Builds upon everything we've learned so far (including today's content):
    - Iterating over sequences (strings, lists, ranges) as well as conditionals
  - More "moving pieces" than Lab 2
  - *Please come* to help hours if you have questions (or to say hi!)
  - **Prelab** due at the beginning of lab
- **HW 3** due tonight at 10 pm on Glow

Do You Have Any Questions?

Last Time

- Introduce iteration using **for loops** to iterate over **sequences**
- Discussed sequence indexing using [ ] and using the `len()` function
- Introduce a new data type (which is also a sequence):
    - `list`

Today's Plan

- Learn more about sequences
  - `in` operator
  - sequence "slicing"
- Iterating over and "accumulating" using lists
- New sequence type: [`range`]

Sequences in Python: Strings

- **Sequences** in Python represent **ordered collections of elements**: e.g., strings, lists, ranges, etc.
- A **string** is an ordered sequences of individual characters
  - Example: `word = "Hello"`
- A **`list`** is a comma-separated, ordered sequence of values
  - Example: `num_list = [1, 5, 8, 9, 15, 27]`
- In CS, we use **zero-indexing**, so we say that `'H'` is at **index** 0 of `word`, `8` is at **index** 2 of `num_list`, and so on
- We can access each character of a sequence using **indices**

```
>>> word[1]            >>> num_list[4]
'e'                    15
```

Slicing Sequences

- We can **extract subsequences** of a sequence using the **slicing** operator `[:]`
- For a given sequence `var`,

```
var[start:end]
```

returns a new sequence of the same type that contains the elements starting at index '`start`' (*inclusive*) and ending at index '`end`' (*exclusive*)

```
>>> vowels = 'aeiou'
>>> vowels[0:2]
'ae'
>>> numList = [2, 4, 8, 16]
>>> numList = [0:-1] # everything except last
[2, 4, 8]
```

Slicing Sequences

- We can **extract subsequences** of a sequence using the **slicing** operator `[:]`
- For a given sequence `var`,

  `var[start:end:step]`

  returns a new sequence of the same type that contains the elements starting at index '`start`' (*inclusive*), ending at index '`end`' (*exclusive*), and using an **(optional)** increment of '`step`'

- By default (if not specified):
  - start defaults to 0 (the beginning of string)
  - end defaults to `len(var)` (end of string)
  - step defaults to `+1`

Examples

```
>>> evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> evens[0:5]
[2, 4, 6, 8, 10]
>>> evens[:8:2]
[2, 6, 10, 14]
>>> evens[::2]
[2, 6, 10, 14, 18]
```

- **Question**. How would we reverse a sequence using slicing?

```
>>> name = "Ephelia"
>>> name[::-1]
'ailehpE
```

Testing Membership: `in` Operator

- The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns `True` or `False`

```
>>> "Williams" in "Williamstown"
True
```

```
>>> "w" in "Williams" # capitalization matters
False
```

```
>>> dog_list = ["Wally", "Velma", "Pixel", "Linus"]
>>> "Linus" in dog_list
True
>>> “Artie" in dog_list
False
```

Testing Membership: `not in` Operator

- The `not in` operator does the opposite of `in`

```
var not in seq
```

"preferred" way (and more readable)

same as

```
not var in seq
```

Summary: Sequence Operations

Operation	Result
`seq[i]`	The `i`'th item of `seq`, when starting with 0
`seq[si:ee]`	slice of `seq` from `si` to `ee`
`seq[si:ee:s]`	slice of `seq` from `si` to `ee` with step `s`
`len(seq)`	length of `seq`
`seq1 + seq2`	The concatenation of `seq1` and `seq2`
`x in seq`	True if `x` is contained within `seq`
`x not in seq`	False if `x` is contained within `seq`

All of these operators work on both **strings** and **lists**!

Exercise: `palindromes`

- A palindrome is a string that is the same forwards and backwards
- The following strings are all examples of palindromes:
  - “”   (any string with length 0)
  - “x”   (any string with length 1)
  - “aba”
  - “racecar”
- The following strings are not palindromes:
- “aA”   (Case mismatch)
- “12321 ” (Un-matched space “ “ at end of string)

Exercise: `palindromes`

- Write a function that iterates over a given list of strings `s_list`, returns a (new) list containing all the strings in `s_list` that are the same forward and backwards (ignoring case).

```
>>> palindromes(["anna", "banana", "kayak", "rigor", "tacit", "hope"])
['anna', 'kayak']
>>> palindromes(["1313", "1110111", "0101"])
['1110111']
>>> palindromes(["level", "stick", "gag"])
['level', 'gag']
```

Exercise: `palindromes`

What is our high level algorithm, in words?

- Go through each word in `s_list`. If the word is a palindrome, append it to our "solution list". After reaching the end of our list, our "solution list" should contain all of the palindromes.

+=

for loop

function that returns a boolean

"accumulator" variable

conditional

Solution: `palindromes`

```
def palindromes(s_list):
    '''Takes a list of string s_list and returns a new list
     containing strings from s_list that
     are the same forwards and backwards'''

    solution = [] # initialize the accumulation variable

    # iterate over each item in seq
    for item in s_list:
        # check if it's a palindrome
        if is_palindrome(item):
            # append to accumulation variable
            solution += [item]

    # return what we accumulated
    return solution
```

How do we implement `is_palindrome(word)`?

`is_palindrome(word)`

What is our high level algorithm, in words?

- Multiple correct algorithms exist!
  - Return true if word is equal to a reversed copy of word
    - **`return word == word[::-1]`**
  - Return true if the first character is equal to the last character AND the second character is equal to the second to last character AND the third character is equal to the third to last character AND …
    - How do we write code that handles arbitrarily long strings?
      - We want a loop that runs **`len(word) // 2`** times because we want to compare **`len(word) // 2`** pairs of characters

Ranges

Ranges (another sequence!)

- Python provides an easy way to iterate over numerical sequences using the **range** data type, which is **another sequence**
- When the `range()` function is given two integer arguments, it returns a ***range object*** of all integers starting at the first and up to, *but not including*, the second (note: default starting value is 0)
- To see the values included in the range, we can pass our range to the `list()` function which returns a **list** of them

```
>>> range(0, 10)
range(0, 10)
```

```
>>> type(range(0, 10))
range
```

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ranges (another sequence!)

- Python provides an easy way to iterate over numerical sequences using the **range** data type, which is **another sequence**
- When the **`range()`** function is given two integer arguments, it returns a ***range object*** of all integers starting at the first and up to, *but not including*, the second (note: default starting value is 0)
- To see the values included in the range, we can … `list()` function which returns a **list** of them

To see elements in range, pass range to `list()` function

```
>>> range(0, 10)
range(0, 10)
```

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> type(range(0, 10))
range
```

A range is a **type** of sequence in Python (like string and list)

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

First argument omitted, **defaults to 0**

Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
```

- In addition to iterating over strings and lists, we can use a **for loop** and a range to simply **repeat** a task.
- This loop print a pattern to the screen.

Looks a lot like `[0, 1, 2, 3, 4]`

```
         i = 0
$        i = 1
$$       i = 2
$$$      i = 3
$$$$     i = 4
```

Using Range For Parallel Iteration

- This also a really convenient way for iterating over two lists in parallel
- Say we wanted to iterate over two lists
- `chars = ['a', 'b', 'c']` and `nums = [1, 2, 3]`
- And form a new list `['a1', 'b2', 'c3']`
- Here's how we'd do it

```
chars = ['a', 'b', 'c']
nums = [1, 2, 3]
# initialize accumulation variable

# for each item in chars
    # add current char to matching num
    # accumulate in a list
```

```
>>> char_nums
['a1', 'b2', 'c3']
```

Using Range For Parallel Iteration

- This also a really convenient way for iterating over two lists in parallel
- Say we wanted to iterate over two lists
- `chars = ['a', 'b', 'c']` and `nums = [1, 2, 3]`
- And form a new list `['a1', 'b2', 'c3']`
- Here's how we'd do it

```
chars = ['a', 'b', 'c']
nums = [1, 2, 3]
char_nums = []

for i in range(0, len(chars)):
    cnum = chars[i] + str(nums[i])
    char_nums += [cnum]
```

Loop Variable

Accumulator Variable

```
>>> char_nums
['a1', 'b2', 'c3']
```

Using **`range`** to check **`palindromes`**

```
def is_palindrome_range(string) :
    # Since we need to compare each char in the "first half"
    # to corresponding char in the "second half",
    # we need to execute len(string) // 2 comparisons
    for i in range(len(string) // 2) :
        if string[i] != string[-(i+1)] :
            return False
    return True
```

Loops: Take-Aways

- `for`..Loops allow us to look at each element in a sequence
    - The **loop variable** defines what the name of that element will be in the loop
    - An optional **accumulator variable** is useful for keeping a running tally of properties of interest
    - Indentation works the same as with if--statements: if it's indented under the loop, it's executed as part of the loop

Importing Functions vs Running as a Script

- **Question.** If you only have function definitions in a file `funcs.py`, and run it as a script, what happens?
  `% python3 funcs.py`

- For testing functions, we want to call /invoke them on various test cases, in Labs, we do this in a separate file called `runtests.py`
  - To add function calls in `runtests.py`, we put them inside the guarded block `if __name__ == "__main__":`
- The statements within this special guarded are only run when the file is run as a ***script*** but not when it is imported as a ***module***
- Let's see an example

```python
# foo.py
# test the role of __name__ variable
print("__name__ is set to", __name__)
```

Running foo.py as a **script**

```
shikhasingh@Shikhas-iMac cs134 % python3 foo.py
__name__ is set to __main__

shikhasingh@Shikhas-iMac cs134 % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021,
14:59:20) [Clang 12.0.5 (clang-1205.0.22.11)] on
darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> import foo
__name__ is set to foo
```

Importing it as a **module**

Takeaway: `if __name__ == "__main__"`

- If you want some statements (like test calls) to be run **ONLY when the file is run as a script**
  - Put them inside the guarded `if __name__ == "__main__"` block

- When we run our automatic tests on your functions we **import them** and this means name is NOT set to main
  - So nothing inside the guarded `if __name__ == "__main__"` block is executed
- This way your testing /debugging statements do not get in the way