Programming Principles in Python (CSCI 503/490)

Sequences

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

For Loop

- for loops in Python are really **for-each** loops
- Always an element that is the current element
  - Can be used to iterate through iterables (containers, generators, strings)
  - Can be used for counting
- ```
  for i in range(5):
      print(i) # 0 1 2 3 4
  ```
- `range(5)` **generates** the numbers 0,1,2,3,4
  ```

Range

- Different method signatures
  - `range(n)` → `0, 1, …, n-1`
  - `range(start, end)` → `start, start + 1, …, end - 1`
  - `range(start, end, step)`
    → `start, start + step, … < end`
- Negative steps:
  - `range(0,4,-1) # <nothing>`
  - `range(4,0,-1) # 4 3 2 1`
- Floating-point arguments are **not** allowed

Sequences

- Strings `"abcde"`, Lists `[1, 2, 3, 4, 5]`, and Tuples `(1, 2, 3, 4, 5)`

- Defining a list: `my_list = [0, 1, 2, 3, 4]`
- But lists can store different types:
  - `my_list = [0, "a", 1.34]`
- Including other lists:
  - `my_list = [0, "a", 1.34, [1, 2, 3]]`
- Others are similar: tuples use parenthesis, strings are delineated by quotes (single or double)

Sequence Operations

- Concatenate: `[1, 2] + [3, 4] # [1,2,3,4]`
- Repeat: `[1,2] * 3 # [1,2,1,2,1,2]`
- Length: `my_list = [1,2]; len(my_list) # 2`

- Concatenate: `(1, 2) + (3, 4) # (1,2,3,4)`
- Repeat: `(1,2) * 3 # (1,2,1,2,1,2)`
- Length: `my_tuple = (1,2); len(my_tuple) # 2`

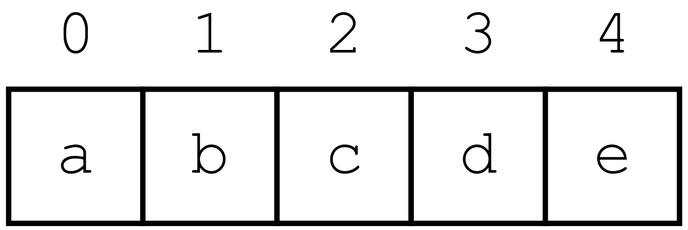
- Concatenate: `"ab" + "cd" # "abcd"`
- Repeat: `"ab" * 3 # "ababab"`
- Length: `my_str = "ab"; len(my_str) # 2`

Assignment 2

- Due Wednesday
- Python control flow and functions
- Compute the 3n+1 function and related values
- Make sure to follow instructions
  - Name the submitted file a2.ipynb
  - Put your name and z-id in the first cell
  - Label each part of the assignment using markdown
  - Make sure to produce output according to specifications

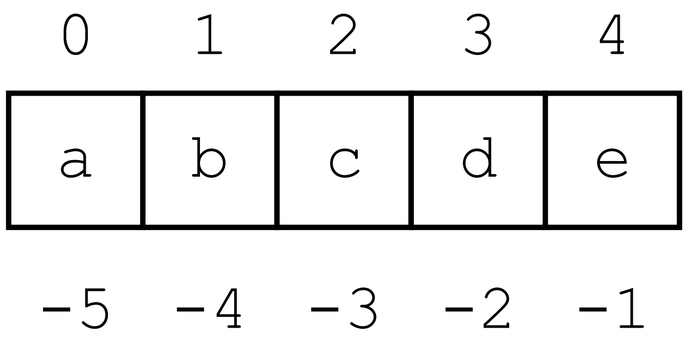
Sequence Indexing

- Square brackets are used to pull out an element of a sequence
- We always start counting at **zero**!
- `my_str = "abcde"; my_str[0] # "a"`
- `my_list = [1,2,3,4,5]; my_list[2] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[5] # IndexError`



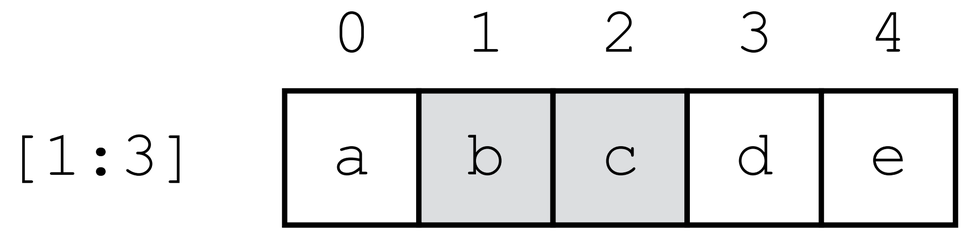
Negative Indexing

- Subtract from the end of the sequence to the beginning
- We always start counting at ~~zero~~ -1 (zero would be ambiguous!)
- `my_str = "abcde"; my_str[-1] # "e"`
- `my_list = [1,2,3,4,5]; my_list[-3] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[-5] # 1`



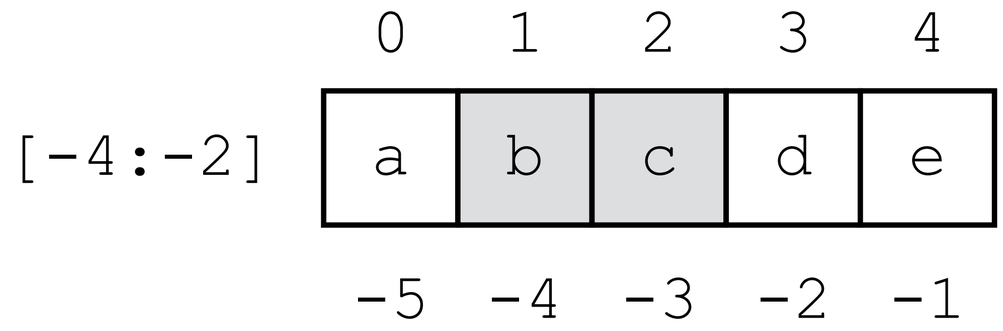
Slicing

- Want a subsequence of the given sequence
- Specify the start and the first index not included
- Returns the same type of sequence
- `my_str = "abcde"; my_str[1:3] # "bc"`
- `my_list = [1,2,3,4,5]; my_list[3:4] # [4]`
- `my_tuple = (1,2,3,4,5); my_tuple[2:99] # (3,4,5)`



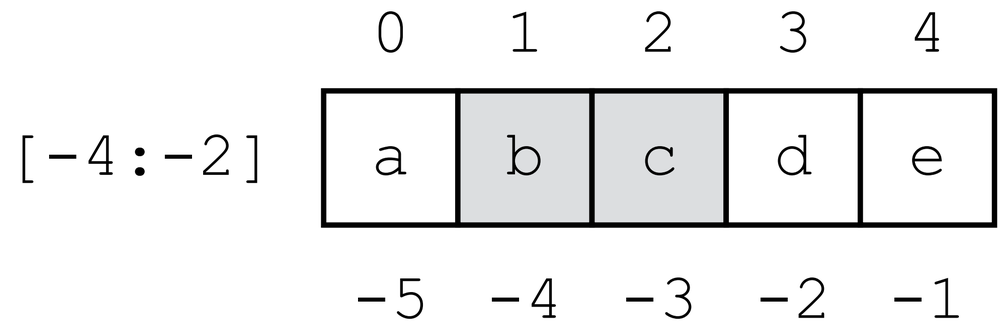
Negative Indices with Slices

- Negative indices can be used instead or with non-negative indices
- `my_str = "abcde"; my_str[-4:-2] # "bc"`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`
- How do we include the last element?
- `my_tuple = (1,2,3,4,5); my_tuple[-2:?]`



Negative Indices with Slices

- Negative indices can be used instead or with non-negative indices
- `my_str = "abcde"; my_str[-4:-2] # "bc"`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`
- How do we include the last element?
- `my_tuple = (1,2,3,4,5); my_tuple[-2:?]`



Implicit Indices

- Don't need to write indices for the beginning or end of a sequence
- Omitting the first number of a slice means start from the beginning
- Omitting the last number of a slice means go through the end
- `my_tuple = (1,2,3,4,5); ~~my_tuple[-2:len(my_tuple)]~~`
- `my_tuple = (1,2,3,4,5); my_tuple[-2:] # (4,5)`
- Can create a **copy** of a sequence by omitting both
- `my_list = [1,2,3,4,5]; my_list[:] # [1,2,3,4,5]`

Iteration

- `for d in sequence:`
  `    # do stuff`
- **Important**: `d` is a **data** item, **not** an **index**!
- `sequence = "abcdef"`
  `for d in sequence:`
  `    print(d, end=" ")  # a b c d e f`
- `sequence = [1,2,3,4,5]`
  `for d in sequence:`
  `    print(d, end=" ")  # 1 2 3 4 5`
- `sequence = (1,2,3,4,5)`
  `for d in sequence:`
  `    print(d, end=" ")  # 1 2 3 4 5`

Membership

- `<expr> in <seq>`
- Returns `True` if the expression is in the sequence, `False` otherwise
- `"a" in "abcde" # True`
- `0 in [1,2,3,4,5] # False`
- `3 in (3, 3, 3, 3) # True`

Sequence Operations

Operator	Meaning
`<seq> + <seq>`	Concatenation
`<seq> * <int-expr>`	Repetition
`<seq>[<int-expr>]`	Indexing
`len(<seq>)`	Length
`<seq>[<int-expr?>:<int-expr?>]`	Slicing
`for <var> in <seq>:`	Iteration
`<expr> in <seq>`	Membership (Boolean)

Sequence Operations

Operator	Meaning
`<seq> + <seq>`	Concatenation
`<seq> * <int-expr>`	Repetition
`<seq>[<int-expr>]`	Indexing
`len(<seq>)`	Length
`<seq>[<int-expr?>:<int-expr?>]`	Slicing
`for <var> in <seq>:`	Iteration
`<expr> in <seq>`	Membership (Boolean)

`<int-expr?>`: may be `<int-expr>` but also can be empty

What's the difference between the sequences?

- Strings can only store characters, lists & tuples can store arbitrary values
- Mutability: strings and tuples are **immutable**, lists are **mutable**
- ```
  my_list = [1, 2, 3, 4]
  my_list[2] = 300
  my_list # [1, 2, 300, 4]
  ```
- `my_tuple = (1, 2, 3, 4); my_tuple[2] = 300 # TypeError`
- `my_str = "abcdef"; my_str[0] = "z" # TypeError`
  ```

List methods

Method	Meaning
`<list>.append(d)`	Add element `d` to end of list.
`<list>.extend(s)`	Add **all** elements in `s` to end of list.
`<list>.insert(i, d)`	Insert `d` into list at index i.
`<list>.pop(i)`	Deletes `i`th element of the list and returns its value.
`<list>.sort()`	Sort the list.
`<list>.reverse()`	Reverse the list.
`<list>.remove(d)`	Deletes first occurrence of `d` in list.
`<list>.index(d)`	Returns index of first occurrence of `d`.
`<list>.count(d)`	Returns the number of occurrences of `d` in list.

List methods

Method	Meaning	Mutate
`<list>.append(d)`	Add element `d` to end of list.	
`<list>.extend(s)`	Add **all** elements in `s` to end of list.	
`<list>.insert(i, d)`	Insert `d` into list at index i.	
`<list>.pop(i)`	Deletes `i`th element of the list and returns its value.	
`<list>.sort()`	Sort the list.	
`<list>.reverse()`	Reverse the list.	
`<list>.remove(d)`	Deletes first occurrence of `d` in list.	
`<list>.index(d)`	Returns index of first occurrence of `d`.	
`<list>.count(d)`	Returns the number of occurrences of `d` in list.	

The del statement

- `pop` works well for removing an element by index plus it **returns** the element
- Can also remove an element at index `i` using
  - `del my_list[i]`
- Note this is very different syntax so I prefer `pop`
- But del can **delete slices**
  - `del my_list[i:j]`
- Also, can delete **identifier** names completely
  -
```
a = 32
del a
a # NameError
```
- This is different than `a = None`

Updating collections

- There are three ways to deal with operations that update collections:
  - Returns an updated **copy** of the list
  - Updates the collection **in place**
  - Updates the collection in place **and returns it**
- `list.sort` and `list.reverse` work **in place** and **don't return** the list
- Common error:
  - `sorted_list = my_list.sort() # sorted_list = None`
- Instead:
  - `sorted_list = sorted(my_list)`

sorted and reversed

- For both sort and reverse, have `sorted` & `reversed` which are **not** in place
- Called with the sequence as the argument
- ```
  my_list = [7, 3, 2, 5, 1]
  for d in sorted(my_list):
      print(d, end=" ")                    # 1 2 3 5 7
  ```
- ```
  my_list = [7, 3, 2, 5, 1]
  for d in reversed(my_list):
      print(d, end=" ")                    # 1 5 2 3 7
  ```
- But this doesn't work:
  - `reversed_list = reversed(my_list)`
- If you need a new list (same as with `range`):
  - `reversed_list = list(reversed(my_list))`

Reversed sort

- Both sort and sorted have a boolean parameter `reverse` that will sort the list in reverse
- ```
  my_list = [7, 3, 2, 5, 1]
  my_list.sort(reverse=True) # my_list now [7, 5, 3, 2, 1]
  ```
- ```
  for i in sorted(my_list, reverse=True):
      print(i, end = " ")    # prints 7 5 3 2 1
  ```
- There is also a `key` parameter that should be a **function** that will be called on each element before comparisons—the outputs will be used to sort
  - Example: convert to lowercase

Nested Sort

- By default, sorts by comparing inner elements in order
- `sorted([[4,2],[1,5],[1,3],[3,5]])`
  - 1st element: `1 == 1 < 3 < 4`
  - 2nd element for equal: `3 < 5`
  - Result: `[[1,3],[1,5],[3,5],[4,2]]`
- Longer lists after shorter lists:
  - `sorted([[1,2],[1]]) # [[1],[1,2]]`

enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
- ```
  for i, d in enumerate(my_list):
      print("index:", i, "element:", d)
  ```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
- ```
  for t in enumerate(my_list):
      i = t[0]
      d = t[1]
      print("index:", i, "element:", d)
  ```

enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
-
```
for i, d in enumerate(my_list):
    print("index:", i, "element:", d)
```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
-
```
for t in enumerate(my_list):
    i = t[0]
    d = t[1]
    print("index:", i, "element:", d)
```