Programming Principles in Python (CSCI 503)

Sequences

Dr. David Koop
Functions

- def <function-name>(<parameter-names>):
  # do stuff
  return res

- Use return to return a value

- Can return more than one value using commas
  
  - def <function-name>(<parameter-names>):
    # do stuff
    return res1, res2

- Use **simultaneous assignment** when calling:
  
  - a, b = do_something(1,2,5)

- If there is no return value, the function returns **None** (a special value)
Scope

- The **scope** of a variable refers to where in a program it can be referenced
- Python has three scopes:
  - **global**: defined outside a function
  - **local**: in a function, only valid in the function
  - **nonlocal**: can be used with nested functions
- Python allows variables in different scopes to have the **same name**
Local Scope

- def f(): # no arguments
  x = 2
  print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)

- Output:
  - x in function: 2
    x in main: 1

- Here, the x in f is in the local scope
Global Keyword for Global Scope

- def f(): # no arguments
  global x
  x = 2
  print("x in function:", x)
  
x = 1
f()
print("x in main:", x)

- Output:
  - x in function: 2
  x in main: 2

- Here, the x in f is in the global scope because of the global declaration
Python as Pass-by-Value?

• `def change_list(inner_list):
    inner_list = [10, 9, 8, 7, 6]

    outer_list = [0, 1, 2, 3, 4]
    change_list(outer_list)
    outer_list # [0, 1, 2, 3, 4]

• Looks like pass by value!
Python as Pass-by-Reference?

- def change_list(inner_list):
  inner_list.append(5)

  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4,5]

- Looks like pass by reference!
Python is Pass-by-object-reference

- AKA passing object references by value
- Python doesn't allocate space for a variable, it just links identifier to a value
- **Mutability** of the object determines whether other references see the change
- Any immutable object will act like pass by value
- Any mutable object acts like pass by reference unless it is reassigned to a new value
Default Parameter Values

• Can add =<value> to parameters
• def rectangle_area(width=30, height=20):
  return width * height

• All of these work:
  - rectangle_area() # 600
  - rectangle_area(10) # 200
  - rectangle_area(10,50) # 500

• If the user does not pass an argument for that parameter, the parameter is set to the default value

• Cannot add non-default parameters after a defaulted parameter
  - def rectangle_area(width=30, height)
Keyword Arguments

- Keyword arguments allow someone calling a function to specify exactly which values they wish to specify without specifying all the values.
- This helps with long parameter lists where the caller wants to only change a few arguments from the defaults.
- `def f(alpha=3, beta=4, gamma=1, delta=7, epsilon=8, zeta=2, eta=0.3, theta=0.5, iota=0.24, kappa=0.134):`
  - `# ...`
- `f(beta=12, iota=0.7)`
Assignment 2

• Due Tonight
• Python control flow and functions
• Do not use containers like lists!
• The 7x+-1 function
• 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
Assignment 3

- Coming soon…
Sequences

- Strings are sequences of characters: "abcde"
- Lists are also sequences: [1, 2, 3, 4, 5]
- + Tuples: (1, 2, 3, 4, 5)
Lists

• 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]]`
Lists Tuples

- Defining a tuple: `my_tuple = (0, 1, 2, 3, 4)`
- But tuples can store different types:
  - `my_tuple = (0, "a", 1.34)`
- Including other tuples:
  - `my_tuple = (0, "a", 1.34, (1, 2, 3))`
- How do you define a tuple with **one** element?
Lists Tuples

• Defining a tuple: `my_tuple = (0, 1, 2, 3, 4)`

• But tuples can store different types:
  - `my_tuple = (0, "a", 1.34)`

• Including other tuples:
  - `my_tuple = (0, "a", 1.34, (1, 2, 3))`

• How do you define a tuple with one element?
  - `my_tuple = (1)`  # doesn't work
  - `my_tuple = (1,)`  # add trailing comma
List Operations

- **Not** like vectors or matrices!
- Concatenate: \([1, 2] + [3, 4] \# [1,2,3,4]\)
- Repeat: \([1,2] \ast 3 \# [1,2,1,2,1,2]\)
- Length: \(\text{my_list} = [1,2]; \text{len(my_list)} \# 2\)
List 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
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

```
0 1 2 3 4
a b c d e
-5 -4 -3 -2 -1
```
Slicing

- Want a subsequence of the given sequence
- Specify the start and the first index not included
- Returns the same type of sequence

```python
my_str = "abcde"; my_str[1:3] # ["b", c"
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] # ["b", c]`
- `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:]`

```
[-4:-2]  | abcde |
    0   1  2  3  4
   a   b  c  d  e
-5 -4 -3 -2 -1
```
Negative Indices with Slices

- Negative indices can be used instead or with non-negative indices
- `my_str = "abcde"; my_str[-4:-2] # ["b", c]`
- `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]`
Indexing Quiz

my_list = ['a', 'b', 'c', 'd', 'e']
Indexing Quiz

my_list = ['a', 'b', 'c', 'd', 'e']

my_list[2]; my_list[-3]; my_list[2:3]
Indexing Quiz

my_list = ['a', 'b', 'c', 'd', 'e']

my_list[2]; my_list[-3]; my_list[2:3]

my_list[1:4]; my_list[-4:-1];
my_list[1:-1]
my_list = ["a", "b", "c", "d", "e"]

```
  a b c d e
my_list[2]; my_list[-3]; my_list[2:3]
```

```
  a b c d e
my_list[1:4]; my_list[-4:-1];
my_list[1:-1]
```

```
  a b c d e
my_list[0:4]; my_list[:4];
my_list[-5:-1]
```
Indexing Quiz

```
my_list = ['a', 'b', 'c', 'd', 'e']
```

```
my_list[2]; my_list[-3]; my_list[2:3]
```

```
my_list[1:4]; my_list[-4:-1];
my_list[1:-1]
```

```
my_list[0:4]; my_list[:4];
my_list[-5:-1]
```

```
my_list[3:]; my_list[-2:]
```
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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;seq&gt; + &lt;seq&gt;</code></td>
<td>Concatenation</td>
</tr>
<tr>
<td><code>&lt;seq&gt; * &lt;int-exp&gt;</code></td>
<td>Repetition</td>
</tr>
<tr>
<td><code>&lt;seq&gt;[&lt;int-exp&gt;]</code></td>
<td>Indexing</td>
</tr>
<tr>
<td><code>len(&lt;seq&gt;)</code></td>
<td>Length</td>
</tr>
<tr>
<td><code>&lt;seq&gt;[&lt;int-exp&gt;..&lt;int-exp&gt;]</code></td>
<td>Slicing</td>
</tr>
<tr>
<td><code>for &lt;var&gt; in &lt;seq&gt;</code>:</td>
<td>Iteration</td>
</tr>
<tr>
<td><code>&lt;expr&gt; in &lt;seq&gt;</code></td>
<td>Membership (Boolean)</td>
</tr>
</tbody>
</table>
Sequence Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;seq&gt; + &lt;seq&gt;</code></td>
<td>Concatenation</td>
</tr>
<tr>
<td><code>&lt;seq&gt; * &lt;int-expr&gt;</code></td>
<td>Repetition</td>
</tr>
<tr>
<td><code>&lt;seq&gt;[&lt;int-expr&gt;]</code></td>
<td>Indexing</td>
</tr>
<tr>
<td><code>len(&lt;seq&gt;)</code></td>
<td>Length</td>
</tr>
<tr>
<td><code>for &lt;var&gt; in &lt;seq&gt;:</code></td>
<td>Iteration</td>
</tr>
<tr>
<td><code>&lt;expr&gt; in &lt;seq&gt;</code></td>
<td>Membership (Boolean)</td>
</tr>
</tbody>
</table>

 `<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

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;list&gt;.append(d)</code></td>
<td>Add element (d) to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.extend(s)</code></td>
<td>Add all elements in (s) to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.insert(i, d)</code></td>
<td>Insert (d) into list at index (i).</td>
</tr>
<tr>
<td><code>&lt;list&gt;.pop(i)</code></td>
<td>Deletes (i)th element of the list and returns its value.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.sort()</code></td>
<td>Sort the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.reverse()</code></td>
<td>Reverse the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.remove(d)</code></td>
<td>Deletes first occurrence of (d) in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.index(d)</code></td>
<td>Returns index of first occurrence of (d).</td>
</tr>
<tr>
<td><code>&lt;list&gt;.count(d)</code></td>
<td>Returns the number of occurrences of (d) in list.</td>
</tr>
</tbody>
</table>
## List methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;list&gt;</code>.append(d)</td>
<td>Add element <code>d</code> to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.extend(s)</td>
<td>Add all elements in <code>s</code> to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.insert(i, d)</td>
<td>Insert <code>d</code> into list at index <code>i</code>.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.pop(i)</td>
<td>Deletes <code>i</code>th element of the list and returns its value.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.sort()</td>
<td>Sort the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.reverse()</td>
<td>Reverse the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.remove(d)</td>
<td>Deletes first occurrence of <code>d</code> in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.index(d)</td>
<td>Returns index of first occurrence of <code>d</code>.</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>.count(d)</td>
<td>Returns the number of occurrences of <code>d</code> in list.</td>
</tr>
</tbody>
</table>

Mutate
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 it

• 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]`
  ```python
  for d in sorted(my_list):
    print(d, end=" ")  # 1 2 3 5 7
  ```
- `my_list = [7, 3, 2, 5, 1]`
  ```python
  for d in reversed(my_list):
    print(d, end=" ")  # 1 5 2 3 7
  ```
- But this doesn't work:
  ```python
  - reversed_list = reversed(my_list)
  ```
- If you need a new list (same as with range):
  ```python
  - 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):`
  ```python
  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):`
  ```python
  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)"