Programming Principles in Python (CSCI 503)

Sequences

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
if, else, elif, pass

- if a < 10:
  print("Small")
else:
  if a < 100:
    print("Medium")
  else:
    if a < 1000:
      print("Large")
    else:
      print("X-Large")

- if a < 10:
  print("Small")
elif a < 100:
  print("Medium")
elif a < 1000:
  print("Large")
else:
  print("X-Large")

- Indentation is critical so else-if branches can become unwieldy (elif helps)
- Remember colons and indentation
- pass can be used for an empty block
while, break, continue

- while <boolean expression>:
  <loop-block>

- Condition is checked at the **beginning** and before each repeat

- **break**: immediately exit the current loop

- **continue**: stop loop execution and go back to the **top** of the loop, checking the condition again

- while \(d > 0\):
  
  \[
  a = \text{get\_next\_input()}
  \]

  if \(a > 100\):
    
    break
  
  if \(a < 10\):
    
    continue

  \(d -= a\)
"...I became convinced that the go to statement should be abolished from all 'higher level' programming languages... The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."

[Dijkstra, 1968]
Loop Styles

• Loop-and-a-Half

```
d = get_data()  # priming rd
while check(d):
    # do stuff
    d = get_data()
```

• Infinite-Loop-Break

```
while True:
    d = get_data()
    if check(d):
        break
    # do stuff
```

• Assignment Expression (Walrus)

```
while check(d := get_data):
    # do stuff
```
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 generates the sequences of integers, one at a time
  - range(n) → 0, 1, ..., n-1
  - range(start, n) → start, start + 1, ..., start + (n-1)
  - range(start, n, step)
    → start, start + step, ..., start + (n-1)*step
Assignment 2

• Due Monday
• Python control flow and functions
• Do not use containers like lists (except for the extra credit)!
• Simple FRACTRAN programs
• 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
Functions

• Call a function $f$: $f(3)$ or $f(3, 4)$ or ... depending on number of parameters
• def <function-name>(<parameter-names>):
  """Optional docstring documenting the function"
  <function-body>
• def stands for function definition
• docstring is convention used for documentation
• Remember the **colon** and **indentation**
• Parameter list can be empty: def $f()$: ...
Functions

• Use `return` to return a value

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

• Can return more than one value using commas

```python
• 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)
Default Values & Keyword Arguments

- Can add \texttt{=<value>} to parameters
- \texttt{def rectangle\_area(width=30, height=20):}
  \hspace{1em} \texttt{return width * height}

- All of these work:
  - \texttt{rectangle\_area()} \# 600
  - \texttt{rectangle\_area(10)} \# 200
  - \texttt{rectangle\_area(10,50)} \# 500

- If the user does not pass an argument for that parameter, the parameter is set to the default value
- Can also pass parameters using \texttt{<name>=<value>} (keyword arguments):
  - \texttt{rectangle\_area(height=50)} \# 1500
Return

- As many return statements as you want
- Always end the function and go back to the calling code
- Returns do not need to match one type/structure (generally not a good idea)

```python
def f(a, b):
    if a < 0:
        return -1
    while b > 10:
        b -= a
        if b < 0:
            return "BAD"
    return b
```

D. Koop, CSCI 503/490, Fall 2021
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: 
  \[ [1,2] * 3 \# [1,2,1,2,1,2] \]
- Length: 
  \[ my\_list = [1,2]; \text{len(my\_list)} \# 2 \]
List Sequence Operations

- **Concatenate**: 
  
  \[ [1, 2] + [3, 4] \equiv [1,2,3,4] \]

- **Repeat**: 
  
  \[ [1,2] \times 3 \equiv [1,2,1,2,1,2] \]

- **Length**: 
  
  \[
  \text{my\_list} = [1,2]; \quad \text{len(my\_list)} \equiv 2
  \]

- **Concatenate**: 
  
  \[(1, 2) + (3, 4) \equiv (1,2,3,4)\]

- **Repeat**: 
  
  \[(1,2) \times 3 \equiv (1,2,1,2,1,2)\]

- **Length**: 
  
  \[
  \text{my\_tuple} = (1,2); \quad \text{len(my\_tuple)} \equiv 2
  \]

- **Concatenate**: 
  
  \"ab\" + \"cd\" \equiv \"abcd\"

- **Repeat**: 
  
  \"ab\" \times 3 \equiv \"ababab\"

- **Length**: 
  
  \[
  \text{my\_str} = \"ab\"; \quad \text{len(my\_str)} \equiv 2
  \]
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
- `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]`
Indexing Quiz

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

```
  a b c d e
```

```
  a b c d e
```

```
  a b c d e
```

```
  a b c d e
```

my_list[2]; my_list[-3]; my_list[2:3]
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[1:-1]
Indexing Quiz

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

1. my_list[2]; my_list[-3]; my_list[2:3]
2. my_list[1:4]; my_list[-4:-1]; my_list[1:-1]
3. my_list[0:4]; my_list[:4]; my_list[-5:-1]
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-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>
## 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**

```python
my_list = [1, 2, 3, 4]
my_list[2] = 300
my_list # [1, 2, 300, 4]
```

```python
my_tuple = (1, 2, 3, 4); my_tuple[2] = 300 # TypeError
```

```python
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 <code>d</code> to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.extend(s)</code></td>
<td>Add <strong>all</strong> elements in <code>s</code> to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.insert(i, d)</code></td>
<td>Insert <code>d</code> into list at index <code>i</code>.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.pop(i)</code></td>
<td>Deletes <code>i</code>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 <code>d</code> in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.index(d)</code></td>
<td>Returns index of first occurrence of <code>d</code>.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.count(d)</code></td>
<td>Returns the number of occurrences of <code>d</code> in list.</td>
</tr>
</tbody>
</table>
## List methods

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