Programming Principles in Python (CSCI 503/490)

Sequences

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Quiz

1. Which is an **invalid** string?
   (a) 'It\'s a dog\'s life'
   (b) "It's a dog's life"
   (c) 'It's a dog's life'
   (d) '''It's a dog's life'''
2. Which is not a valid python identifier?
   (a) _in_order_list
   (b) mañana
   (c) str
   (d) 3bears
Quiz

3. What type of statement did Dijkstra “consider harmful”?
   (a) goto
   (b) continue
   (c) break
   (d) do-while
Quiz

4. Which expression computes whether \( a \) is greater than 2 and \( b \) is not equal to 10000?
(a) \( \text{not } a \leq 2 \text{ and } b \neq 10000 \)
(b) \( a > 2 \text{ and } b 
eq 10000 \)
(c) \( \neg (a \leq 2 \text{ or } b = 10000) \)
(d) \( a > 2 \text{ and } b \text{ is not } 10000 \)
Quiz

5. What does $4 + 9 / 2 - 2$ evaluate to?
   (a) 6.5
   (b) 4
   (c) 6
   (d) 4.5
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 = get_next_input()  
  if a > 100:  
    break  
  if a < 10:  
    continue  
  d -= a
For a number of years I have been familiar with the observation 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.
Loop Styles

• Loop-and-a-Half

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

• Infinite-Loop-Break

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

• Assignment Expression (Walrus)

```python
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(5) generates the numbers 0,1,2,3,4
Range

- Python has lists which allow enumeration of all possibilities: [0,1,2,3,4]
- Can use these in for loops
  - for i in [0,1,2,3,4]:
    - print(i) # 0 1 2 3 4
- but this is less efficient than range (which is a generator)
  - for i in range(5):
    - print(i) # 0 1 2 3 4
- List must be stored, range doesn't require storage
- Printing a range doesn't work as expected:
  - print(range(5)) # prints "range(0, 5)"
  - print(list(range(5)) # prints "[0, 1, 2, 3, 4]"
Looping Errors

- # for loop - summing the numbers 1 to 10
  
  ```python
  n = 10
  cur_sum = 0
  for i in range(n):
      cur_sum += i
  
  print("The sum of the numbers from 1 to", n, "is ", cur_sum)
  ```
Assignment 2

- Due next Wednesday
- Python control flow and functions
-
Functions
Functions

• Call a function \( f: f(3) \) or \( f(3,4) \) or ... depending on number of parameters

• \texttt{def <function-name>(<parameter-names>):}
  
  """Optional docstring documenting the function""
  
  \texttt{<function-body>}

• \texttt{def} stands for function definition

• docstring is convention used for documentation

• Remember the \texttt{colon} and \texttt{indentation}

• Parameter list can be empty: \texttt{def f(): ...}
Functions

- Use `return` to return a value
  - `def <function-name>(<parameter-names>):`
    - # do stuff
    - return res

- 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)
Default Values & Keyword Arguments

- 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

- Can also pass parameters using `<name>=<value>` (keyword arguments):
  - `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
```
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: \([1,2] \times 3 \) # \([1,2,1,2,1,2]\)
- Length: my_list = [1,2]; len(my_list) # 2
List Sequence Operations

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

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

- Concatenate: "ab" + "cd" \# "abcd"
- Repeat: "ab" \times 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] # "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:]

```
  0 1 2 3 4
[-4:-2]   a b c d e
-5 -4 -3 -2 -1
```
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

```python
my_tuple = (1, 2, 3, 4, 5);
my_tuple[-2:len(my_tuple)]
```

```python
my_tuple = (1, 2, 3, 4, 5);
my_tuple[-2:]  # (4, 5)
```

- Can create a **copy** of a sequence by omitting both
- ```python
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

<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]
• 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>list.append(d)</code></td>
<td>Add element <code>d</code> to end of list.</td>
</tr>
<tr>
<td><code>list.extend(s)</code></td>
<td>Add all elements in <code>s</code> to end of list.</td>
</tr>
<tr>
<td><code>list.insert(i, d)</code></td>
<td>Insert <code>d</code> into list at index <code>i</code>.</td>
</tr>
<tr>
<td><code>list.pop(i)</code></td>
<td>Deletes <code>i</code>th element of the list and returns its value.</td>
</tr>
<tr>
<td><code>list.sort()</code></td>
<td>Sort the list.</td>
</tr>
<tr>
<td><code>list.reverse()</code></td>
<td>Reverse the list.</td>
</tr>
<tr>
<td><code>list.remove(d)</code></td>
<td>Deletes first occurrence of <code>d</code> in list.</td>
</tr>
<tr>
<td><code>list.index(d)</code></td>
<td>Returns index of first occurrence of <code>d</code>.</td>
</tr>
<tr>
<td><code>list.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>
</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 all 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>