Programming Principles in Python (CSCI 503/490)

Functions

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
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: \([1,2] \times 3 \) \# \([1,2,1,2,1,2]\)
- Length: \(my\_list = [1,2]; \) \(\text{len}(my\_list) \) \# 2

- Concatenate: \((1, 2) + (3, 4) \) \# \((1,2,3,4)\)
- Repeat: \((1,2) \times 3 \) \# \((1,2,1,2,1,2)\)
- Length: \(my\_tuple = (1,2); \) \(\text{len}(my\_tuple) \) \# 2

- Concatenate: \("ab" + "cd" \) \# "abcd"
- Repeat: \("ab" \times 3 \) \# "ababab"
- Length: \(my\_str = "ab"; \) \(\text{len}(my\_str) \) \# 2
my_list = ['a', 'b', 'c', 'd', 'e']
Indexing & Slicing Quiz

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

my_list[3]; my_list[-2]; my_list[3:4]
Indexing & Slicing Quiz

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

my_list[3]; my_list[-2]; my_list[3:4]

my_list[1:3]; my_list[-4:-2];
my_list[1:-2]

my_list[1:-2]
my_list = ['a', 'b', 'c', 'd', 'e']

my_list[3]; my_list[-2]; my_list[3:4]

my_list[1:3]; my_list[-4:-2];
my_list[1:-2]

my_list[0:4]; my_list[:4];
my_list[-5:-1]
Indexing & Slicing Quiz

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

<table>
<thead>
<tr>
<th>a b c d e</th>
<th>my_list[3]; my_list[-2]; my_list[3:4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d e</td>
<td>my_list[1:3]; my_list[-4:-2]; my_list[1:-2]</td>
</tr>
<tr>
<td>a b c d e</td>
<td>my_list[0:4]; my_list[:4]; my_list[-5:-1]</td>
</tr>
<tr>
<td>a b c d e</td>
<td>my_list[3:]; my_list[-2:]</td>
</tr>
</tbody>
</table>
Indexing (Positive and Negative)

- Positive indices start at zero, negative at -1
- `my_str = "abcde"; my_str[1] # "b"
- `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

- Positive or negative indices can be used at any step
- `my_str = "abcde"; my_str[1:3] # ["b", c]`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`

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

```
  0  1  2  3  4
[1:3] a b c d e
[-4:-2] -5 -4 -3 -2 -1
```
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
## 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 d to end of list.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.extend(s)</code></td>
<td>Add all elements in s to end of list.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.insert(i, d)</code></td>
<td>Insert d into list at index i.</td>
<td></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>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.sort()</code></td>
<td>Sort the list.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.reverse()</code></td>
<td>Reverse the list.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.remove(d)</code></td>
<td>Deletes first occurrence of d in list.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.index(d)</code></td>
<td>Returns index of first occurrence of d.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;list&gt;.count(d)</code></td>
<td>Returns the number of occurrences of d in list.</td>
<td></td>
</tr>
</tbody>
</table>
Assignment 2

- Due next Tuesday
- Python control flow and functions
- Do not use containers like lists!
- Compute Compound Interest and Compare Situations
- 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
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):`
  ```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)
  ```
Tuples

- Tuples are **immutable** sequences
- We've actually seen tuples a couple of times already
  - Simultaneous Assignment
  - Returning Multiple Values from a Function
- Python allows us to omit parentheses when it's clear
  - \( b, a = a, b \)  # same as \((b, a) = (a, b)\)
  - \( t1 = a, b \)  # don't normally do this
  - \( c, d = f(2, 5, 8) \)  # same as \((c, d) = f(2, 5, 8)\)
  - \( t2 = f(2, 5, 8) \)  # don't normally do this
Packing and Unpacking

• def f(a, b):
    if a > 3:
        return a, b-a # tuple packing
    return a+b, b # tuple packing
• c, d = f(4, 3) # tuple unpacking

• Make sure to unpack the correct number of variables!
• c, d = a+b, a-b, 2*a # ValueError: too many values to unpack
• Sometimes, check return value before unpacking:
  - retval = f(42)
    if retval is not None:
      c, d = retval
Packing and Unpacking

• def f(a, b):
  if a > 3:
    return a, b-a # tuple packing
  return a+b, b # tuple packing

• c, d = f(4, 3) # tuple unpacking

• Make sure to unpack the correct number of variables!
• c, d = a+b, a-b, 2*a # ValueError: too many values to unpack
• Sometimes, check return value before unpacking:
  - retval = f(42)
    if retval is not None:
      c, d = retval
Packing and Unpacking

- def f(a, b):
  if a > 3:
    return a, b-a # tuple packing
  return a+b, b # tuple packing

- c, d = f(4, 3) # tuple unpacking

- Make sure to unpack the correct number of variables!
- c, d = a+b, a-b, 2*a # ValueError: too many values to unpack
- Sometimes, check return value before unpacking:
  - retval = f(42)
    if retval is not None:
      c, d = retval

D. Koop, CSCI 503/490, Fall 2022
Unpacking other sequences

- You can unpack other sequences, too
  - `a, b = 'ab'`
  - `a, b = ['a', 'b']`

- Why is list unpacking rare?
Other sequence methods

- my_list = [7, 2, 1, 12]
- Math methods:
  - max(my_list) # 12
  - min(my_list) # 1
  - sum(my_list) # 22
- zip: combine two sequences into a single sequence of tuples
  - zip_list = list(zip(my_list, "abcd"))
    zip_list # [(1, 'a'), (2, 'b'), (7, 'c'), (12, 'd')]
  - Use this instead of using indices to count through both
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""
  <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 \texttt{return} to return a value
  
  \begin{verbatim}
  def <function-name>(<parameter-names>):
    # do stuff
    return res
  \end{verbatim}

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

- Use \texttt{simultaneous assignment} when calling:
  
  - \begin{verbatim}
  a, b = do_something(1,2,5)
  \end{verbatim}

- If there is no return value, the function returns \texttt{None} (a special value)
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)
• `def f(a,b):
    if a < 0:
        return -1
    while b > 10:
        b -= a
        if b < 0:
            return "BAD"
    return b`
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**.
Global read

• `def f(): # no arguments
    print("x in function:", x)

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

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

• Here, the `x` in `f` is read from the global scope
Try to modify global?

- **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

• `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
What is the scope of a parameter of a function?
Depends on whether Python is pass-by-value or pass-by-reference
Pass by value

• Detour to C++ land:

    void f(int x) {
        x = 2;
        cout << "Value of x in f: " << x << endl;
    }

    main() {
        int x = 1;
        f(x);
        cout << "Value of x in main: " << x;
    }

Pass by value

- Detour to C++ land:
  
  ```
  void f(int x) {
    x = 2;
    cout << "Value of x in f: " << x << endl;
  }

  main() {
    int x = 1;
    f(x);
    cout << "Value of x in main: " << x;
  }
  ```

  Output:
  Value of x in f: 2
  Value of x in main: 1
Pass by reference

• Detour to C++ land:

```cpp
void f(int & x) {
    x = 2;
    cout << "Value of x in f: " << x << endl;
}

main() {
    int x = 1;
    f(x);
    cout << "Value of x in main: " << x;
}
```
Pass by reference

- Detour to C++ land:

```cpp
void f(int & x) {
    x = 2;
    cout << "Value of x in f: " << x << endl;
}

main() {
    int x = 1;
    f(x);
    cout << "Value of x in main: " << x;
}
```
Pass by reference

- Detour to C++ land:
  - void f(int & x) {
    x = 2;
    cout << "Value of x in f: " << x << endl;
  }

  main() {
    int x = 1;
    f(x);
    cout << "Value of x in main: " << x;
  }

Output:
Value of x in f: 2
Value of x in main: 2
Pass by reference

- Detour to C++ land:
  ```cpp
  void f(int & x) {
    x = 2;
    cout << "Value of x in f: " << x << endl;
  }
  
  main() {
    int x = 1;
    f(x);
    cout << "Value of x in main: " << x;
  }
  
  Output:
  Value of x in f: 2
  Value of x in main: 2
  ```
Is Python pass-by-value or pass-by-reference?
Neither
Example 1

• 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!
Example 2

- `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!
What's going on?
Think about how assignment works in Python
Different than C++
Example 1

• 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]
Example 1

• 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]
Example 1

- def change_list(inner_list):
  \[
  \text{inner_list} = [10,9,8,7,6]
  \]

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

- `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]`
Example 2

• 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]
Example 2

- **def change_list(inner_list):**
  
  ```python
  inner_list.append(5)
  ```

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

- `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]`
Example 2

- 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]
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
Remember: global allows assignment in functions

• def change_list():
  global a_list
  a_list = [10,9,8,7,6]

  a_list = [0,1,2,3,4]
  change_list()
  a_list # [10,9,8,7,6]
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)
Don't use mutable values as defaults!

- def append_to(element, to=[]):
  to.append(element)
  return to

- my_list = append_to(12)
  my_list # [12]

- my_other_list = append_to(42)
  my_other_list # [12, 42]
Use None as a default instead

- `def append_to(element, to=None):
  if to is None:
    to = []
    to.append(element)
  return to

- `my_list = append_to(12)
  my_list # [12]

- `my_other_list = append_to(42)
  my_other_list # [42]

- If you're not mutating, this isn't an issue