Comprehension

Thomas Schwarz, SJ
Marquette University
Programming Styles

• Styles of Programming

  • Imperative Programming:
    • Describe in detail how computation proceeds
    • Basically, change states of variables
    • This is what we practiced up till now
Programming Styles

- Functional Programming
  - Define functions
    - Specify program behavior by executing nested functions
    - Pure functional programming: No variables that capture a state
  - Advantage: Easier to prove programming correctness
Programming Styles

- Declarative Programming
  - Specify what a program should do
    - System figures out how to do it.
  - Example 1: Prolog (Classic AI programming language)
    - Specify rules in Prolog:
      - `animal(X) :- cat(X)` means every cat is an animal
      - `?- cat(tom).` means that tom is a cat
    - You can ask about the world defined by these rules
      - `?- animal(X).` asks for what things are animals
    - Prolog consists of rules and base facts, then on its own finds out other facts.
Programming Styles

• Declarative Programming:
  • Example 2: SQL — Database Language
    • Database consists of relations stored in various tables
  • Example:

<table>
<thead>
<tr>
<th>Marquette_ID</th>
<th>First_Name</th>
<th>Family_Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123123007</td>
<td>David</td>
<td>Roy</td>
<td>1984 31st Street, Milwaukee, WI 54321</td>
</tr>
<tr>
<td>97007007</td>
<td>Thomas</td>
<td>Schwarz</td>
<td>4821 Wisconsin Ave, Milwaukee, WI 54213</td>
</tr>
<tr>
<td>14309873</td>
<td>Joseph</td>
<td>Cuelho</td>
<td>9821 12th Avenue, Milwaukee, WI 54321</td>
</tr>
<tr>
<td>90874132</td>
<td>Donald</td>
<td>Drumpf</td>
<td>321 Pennsylvania Ave, Madison, WI 32451</td>
</tr>
</tbody>
</table>
Programming Styles

• Declarative Programming:

• Example SQL:

  • SQL statement describes all combinations of record pieces

    SELECT first_name, family_name FROM addresses, classes

    WHERE classes.name = "COSC1010" and classes.role = "instructor" and classes.id = addresses.id
Programming Styles

• Declarative Programming:
  • Example SQL:
    • SQL statement describes all combinations of record pieces
    • How the database engine performs the query is not specified
    • In fact, for complicated queries, the database will try out several ways before selecting the actual algorithms
Programming Styles

• Object-Oriented Programming
  • Program defined various objects
    • Objects have data and methods
      • E.g. Marquette Persons have IDs, names, addresses, …
    • Classes have lists of participants
  • We will learn Object-Oriented (OO) programming in this class
Comprehension

• List comprehension is used in functional programming but it becomes handy
  
  • We define a list with a for clause within the brackets that define the list.
  
  • Here are two ways to construct a list consisting of squares

\[
\text{lista} = []
\text{for } i \text{ in range}(100): \quad \text{lista.append}(i**2)
\]

\[
\text{lista} = [i**2 \text{ for } i \text{ in range}(100)]
\]
Comprehension

\[
[ \text{x}**2 \text{ for } \text{x} \text{ in range}(100) ]
\]
Self Test

- The following code fragment defines a list of elements
- Use list comprehension in order to generate the same list

- Use the interactive window in IDLE

```python
>>> lista = []
>>> for i in range(10):
    lista.append(i**3-i**2+i-1)

>>> lista
[-1, 0, 5, 20, 51, 104, 185, 300, 455, 656]
```
Self Test

Pause the presentation until you have solved the problem
Self Test Solution

```python
>>> lista = [i**3-i**2+i-1 for i in range(10)]
>>> lista
[-1, 0, 5, 20, 51, 104, 185, 300, 455, 656]
```
Comprehension

- List comprehension can add an if-condition

\[
[ x**2 \text{ for } x \text{ in range}(100) \text{ if } x \% 2 == 0 ]
\]

- Result is now all even squares.
Comprehension

• List comprehension can be quite involved
  • Remember that we can check for types of variables
  • We use the built-in function `isinstance( )`
  • Example: `isinstance(345, int)` is True
  • Application to list comprehension: Squaring the elements of a list (a_list) that are integers

```
>>> a_list = [1, "4", 9, "a", 0, 4]
>>> [e**2 for e in a_list if isinstance(e, int)]
[1, 81, 0, 16]
```
Comprehension

• We can nest comprehensions

• A list of all composite numbers between 2 and 100.
  
  • A composite number is a product of two integers $i$ and $j$
    that are larger than 1.

  $[i\times j \text{ for } i \text{ in } \text{range}(2,51) \text{ for } j \text{ in } \text{range}(2,101) \text{ if } i\times j < 100]$  

• However, the result contains many repeated numbers
Comprehensions

- Luckily, we can use a set instead:

\{i*j \text{ for } i \text{ in range(2,51) for } j \text{ in range(2,51) if } i*j < 100\}

- The difference is just curly brackets instead of rectangular brackets

- The result is now simpler:

{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33, 34, 35, 36, 38, 39, 40, 42, 44, 45, 46, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58, 60, 62, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 76, 77, 78, 80, 81, 82, 84, 85, 86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 98, 99}
Comprehensions

• We can now get all of the prime numbers between 2 and 100 by using this set, using comprehension on top of comprehension

\{i \text{ for } i \text{ in range}(2,100) \text{ if } i \text{ not in } \{i*j \text{ for } i \text{ in range}(2,51) \text{ for } j \text{ in range}(2,51) \text{ if } i*j < 100\}\}

• This is cool but will not win any price for clarity

• You can make it more comprehensible if you define a set of composite numbers before using it
Self Test

• Use the previous example to generate a set of all numbers between 1 and 100 (included) that are not squares
Self Test Solution

```python
seta = {i for i in range(1,101) if i not in {i*i for i in range(10)}}
```
Comprehensions

• You can also use comprehension on dictionaries

• Here is how you create a dictionary that associates integers up to 100*100 to their square root

  • \{i*i: i for i in range(101)}

```python
>>> {i*i: i for i in range(101)}
```
Comprehensions

- And here is how you can try to “invert” a dictionary where the roles of keys and values are swapped
  
  \[
  \text{drev} = \{d[\text{key}]:\text{key} \text{ for key in } d\}
  \]

- This one works well, because the values are different for different keys

  \[
  \begin{align*}
  &\text{d} = \{1:4, 2:5, 3:7, 4:8, 5:9\} \\
  &\text{drev} = \{d[\text{key}]:\text{key} \text{ for key in } d\} \\
  &\{4: 1, 5: 2, 7: 3, 8: 4, 9: 5\}
  \end{align*}
  \]

- And this one inverts with some arbitrariness

  \[
  \begin{align*}
  &\text{d} = \{1:4, 2:5, 3:4, 4:5, 6:7, 7:6\} \\
  &\text{drev} = \{d[\text{key}]:\text{key} \text{ for key in } d\} \\
  &\{4: 3, 5: 4, 7: 6, 6: 7\}
  \end{align*}
  \]
Self Test

• You are given a function `func` that takes one integer argument

• You want to create a memoization dictionary that associates `i for i in range(100)` with `func(i)`
Self Test Answer

mem_func = {i: func(i) for i in range(101)}

func = lambda x: 3*x+4

gives

```python
>>> func = lambda x: 3*x+4
>>> mem = {x: func(x) for x in range(101)}
>>> mem
```
Map, Filter
Map

- Map allows you to apply a function to all elements of a list.
- Example:

  ```python
  func = lambda x: x+3
  list(map(func, [2,3,4]))
  ```

- Why the list? map returns an iterator (so that it does not waste memory on values that are not used).

  ```python
  >>> func = lambda x: x+3
  >>> list(map(func, [2,3,4]))
  [5, 6, 7]
  ```
Filter

- You filter a list by applying a condition
- The result is the list formed by all elements that satisfy the condition
  - You need to have a boolean function, i.e. a function that returns True or False
  - Here is an example of such a function:

\[
\text{lambda } x: x \% 2 == 0
\]

- Returns True if \( x \) is divisible by 2
- Returns False otherwise
- \( x \% 2 \) is zero if and only if \( x \) is even
Filter

- The function `filter(function, sequence)` return an iterable of all elements in the sequence `t` that render the function `True`.

```python
>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 44, 65, 109, 174, 283]
>>> list(filter(lambda x: x%2==0, fibonacci))
[0, 2, 8, 44, 174]
>>> list(filter(lambda x: x%2==1, fibonacci))
[1, 1, 3, 5, 13, 21, 65, 109, 283]
```