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:

Marquette_ID	First_Name	Family_Name	Address
123123007	David	Roy	1984 31st Street, Milwaukee, WI 54321
97007007	Thomas	Schwarz	4821 Wisconsin Ave, Milwaukee, WI 54213
14309873	Joseph	Cuelho	9821 12th Avenue, Milwaukee, WI 54321
90874132	Donald	Drumpf	321 Pennsylvania Ave, Madison, WI 32451

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

```
lista = []
for i in range(100):
    lista.append(i**2)
```

```
lista = [i**2 for i in range(100)]
```

Comprehension

```
[x**2 for x in range(100)]
```

output expression

generator expression

variable

list
(or list-like expression)

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

```
>>> 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

```
>>> 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 for x in range(100) 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*j for i in range(2,51) for j in range(2,101) if i*j < 100]
```

- However, the result contains many repeated numbers

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

Comprehensions

- Luckily, we can use a set instead:

```
{i*j for i in range(2,51) for j 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 for i in range(2,100) if i not in
{i*j for i in range(2,51) for j in range(2,51) 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

```
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)}`

```
>>> {i*i: i for i in range(101)}
{0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6, 49: 7, 64: 8, 81: 9, 100: 10, 121:
 11, 144: 12, 169: 13, 196: 14, 225: 15, 256: 16, 289: 17, 324: 18, 361: 19, 400
: 20, 441: 21, 484: 22, 529: 23, 576: 24, 625: 25, 676: 26, 729: 27, 784: 28, 84
1: 29, 900: 30, 961: 31, 1024: 32, 1089: 33, 1156: 34, 1225: 35, 1296: 36, 1369:
 37, 1444: 38, 1521: 39, 1600: 40, 1681: 41, 1764: 42, 1849: 43, 1936: 44, 2025:
 45, 2116: 46, 2209: 47, 2304: 48, 2401: 49, 2500: 50, 2601: 51, 2704: 52, 2809:
 53, 2916: 54, 3025: 55, 3136: 56, 3249: 57, 3364: 58, 3481: 59, 3600: 60, 3721:
 61, 3844: 62, 3969: 63, 4096: 64, 4225: 65, 4356: 66, 4489: 67, 4624: 68, 4761:
 69, 4900: 70, 5041: 71, 5184: 72, 5329: 73, 5476: 74, 5625: 75, 5776: 76, 5929:
 77, 6084: 78, 6241: 79, 6400: 80, 6561: 81, 6724: 82, 6889: 83, 7056: 84, 7225:
 85, 7396: 86, 7569: 87, 7744: 88, 7921: 89, 8100: 90, 8281: 91, 8464: 92, 8649:
 93, 8836: 94, 9025: 95, 9216: 96, 9409: 97, 9604: 98, 9801: 99, 10000: 100}
```

Comprehensions

- And here is how you can try to “invert” a dictionary where the roles of keys and values are swapped

```
drev = {d[key]:key for key in d}
```

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

```
>>> d = {1:4, 2:5, 3:7, 4:8, 5:9}
>>> {d[key]:key for key in d}
{4: 1, 5: 2, 7: 3, 8: 4, 9: 5}
```

- And this one inverts with some arbitrariness

```
>>> d = {1:4, 2:5, 3:4, 4:5, 6:7, 7:6}
>>> {d[key]:key for key in d}
{4: 3, 5: 4, 7: 6, 6: 7}
```

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

```
>>> func = lambda x: 3*x+4
>>> mem = {x: func(x) for x in range(101)}
>>> mem
{0: 4, 1: 7, 2: 10, 3: 13, 4: 16, 5: 19, 6: 22, 7: 25, 8: 28, 9: 31, 10: 34, 11:
 37, 12: 40, 13: 43, 14: 46, 15: 49, 16: 52, 17: 55, 18: 58, 19: 61, 20: 64, 21:
 67, 22: 70, 23: 73, 24: 76, 25: 79, 26: 82, 27: 85, 28: 88, 29: 91, 30: 94, 31:
 97, 32: 100, 33: 103, 34: 106, 35: 109, 36: 112, 37: 115, 38: 118, 39: 121, 40:
 124, 41: 127, 42: 130, 43: 133, 44: 136, 45: 139, 46: 142, 47: 145, 48: 148, 49
: 151, 50: 154, 51: 157, 52: 160, 53: 163, 54: 166, 55: 169, 56: 172, 57: 175, 5
8: 178, 59: 181, 60: 184, 61: 187, 62: 190, 63: 193, 64: 196, 65: 199, 66: 202,
67: 205, 68: 208, 69: 211, 70: 214, 71: 217, 72: 220, 73: 223, 74: 226, 75: 229,
 76: 232, 77: 235, 78: 238, 79: 241, 80: 244, 81: 247, 82: 250, 83: 253, 84: 256
, 85: 259, 86: 262, 87: 265, 88: 268, 89: 271, 90: 274, 91: 277, 92: 280, 93: 28
3, 94: 286, 95: 289, 96: 292, 97: 295, 98: 298, 99: 301, 100: 304}
```

Map, Filter

Map

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

```
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)

```
>>> 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:

```
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.

```
>>> 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]
```