Comprehension in Action

Python
Getting the listing of a directory

• Task: Generate a listing of all files in a directory that end in “.py”
  • Tool: import the os module and use listdir

```python
[filename for filename in os.listdir(directoryname) if filename.endswith(".py")]
```
Creating sub-directories

• Task: We want to create a sub-dictionary of a dictionary where the keys are restricted by a condition
  • Use dictionary comprehension

```python
def evenkeys(dictionary):
    return { i:dictionary[i] for i in dictionary if i%2==0}
```
Filtering a list

• We want to filter a list using a criterion

1. We can use the filter function

2. We can use list comprehension, which is often simpler

• Example: Only display the positive elements of this large list

```python
>>> rlist
[20, -1, 3, 0, 17, 1, 20, 19, 24, 4, 21, 0, 4, 7, 20, 2, 1, 13, 0, 21, 23, 6, 2, 22, 4, 3, 6, 2, 13, -5, 3, 13, 20, 23, 14, 13, 13, 20, 10, 24, 9, -1, -4, 22, 1 5, 21, 18, -1, 16, 13, 1, 3, 12, 21, 0, 9, 4, 24, -3, 4, 10, 8, 1, 19, 3, 20, 4, 5, 25, 8, 8, 14, -5, 23, 24, 14, 1, 0, -5, -3, 3, -4, 11, 1, 8, 17, 2, 2, 23, 6 , 2, 25, 15, 4, 23, 20, 5, -3, 11, 16]

>>> list(filter(lambda x: x>0, rlist))
[20, 3, 17, 1, 20, 19, 24, 4, 21, 4, 7, 20, 2, 1, 13, 21, 23, 6, 2, 22, 4, 3, 6, 2, 13, 3, 13, 20, 23, 14, 13, 13, 20, 10, 24, 9, 22, 15, 21, 18, 16, 13, 1, 3, 12, 21, 9, 4, 24, 4, 10, 8, 1, 19, 3, 20, 4, 5, 25, 8, 8, 14, 23, 24, 14, 1, 3, 11, 1, 8, 17, 2, 2, 23, 6, 2, 25, 15, 4, 23, 20, 5, 11, 16]

>>> [x for x in rlist if x>0]
[20, 3, 17, 1, 20, 19, 24, 4, 21, 4, 7, 20, 2, 1, 13, 21, 23, 6, 2, 22, 4, 3, 6, 2, 13, 3, 13, 20, 23, 14, 13, 13, 20, 10, 24, 9, 22, 15, 21, 18, 16, 13, 1, 3, 12, 21, 9, 4, 24, 4, 10, 8, 1, 19, 3, 20, 4, 5, 25, 8, 8, 14, 23, 24, 14, 1, 3, 11, 1, 8, 17, 2, 2, 23, 6, 2, 25, 15, 4, 23, 20, 5, 11, 16]```
Mapping a list

- We want to apply a function to all elements in a list

```python
>>> rlist = [random.randint(-10, 20) for _ in range(20)]
>>> rlist
[-2, -9, 20, -10, -9, 19, -4, 1, 16, 3, 8, -10, 4, -2, 11, 8, 11, -7, -2, -3]
>>> list(map(lambda x: (x-6)**2, rlist))
[64, 225, 196, 256, 225, 169, 100, 25, 100, 9, 4, 256, 4, 64, 25, 4, 25, 169, 64, 81]
>>> [(x-6)**2 for x in rlist]
[64, 225, 196, 256, 225, 169, 100, 25, 100, 9, 4, 256, 4, 64, 25, 4, 25, 169, 64, 81]
```
Zip
Zip

• Often we have related data in a number of lists
  • Example: list of student names, list of grades, list of high school
    • [“Frankieboy”, “Violet”, “Kumar”, “Dshenghis”]
    • [“D”, “A”, “B”, “C”]
    • [“MPS1”, “MH”, “MH”, “MPS59”]

• Zipping will create a zip object that generates the tuples ("Frankieboy", "D", "MPS1"), ("Violet","A","MH"), ("Kumar", "B", "MH"), ("Dshenghis","C", "MPS59")
Zip

- We can reach the same effect with list comprehension, but since we cannot enumerate in parallel through several iterables, we need to use indices.

```python
>>> names = ['Albertina', 'Bertram', 'Chris', 'David']
>>> grades = ['A', 'B', 'C', 'D']
>>> highs = ['MH', 'SHH', 'LGH', 'MHT']
>>> zip(names, grades, highs)
<zip object at 0x1153e8bc8>
>>> list(zip(names, grades, highs))
[('Albertina', 'A', 'MH'), ('Bertram', 'B', 'SHH'), ('Chris', 'C', 'LGH'), ('David', 'D', 'MHT')]
>>> [ (names[i], grades[i], highs[i]) for i in range(len(names))]
[('Albertina', 'A', 'MH'), ('Bertram', 'B', 'SHH'), ('Chris', 'C', 'LGH'), ('David', 'D', 'MHT')]
```
Zip

- What happens if you give zip iterables of different length
  - E.g. a list of 5, a list of 4 and a list of 3 elements?
  - The result is a zip object of length the minimum of the lengths.
Undoing a zip:

If you make a list `alist` out of a zip object, you can break it apart with the `zip(*alist)` command.

```python
>>> names = ['Albertina', 'Bertram', 'Chris', 'David']
>>> grades = ['A', 'B', 'C', 'D']
>>> highschools = ['MH', 'SHH', 'MPSS7', 'LGH']
>>> alist = list(zip(names, grades, highschools))
>>> alist
[('Albertina', 'A', 'MH'), ('Bertram', 'B', 'SHH'), ('Chris', 'C', 'MPSS7'), ('David', 'D', 'LGH')]
>>> list(zip(*alist))
[('Albertina', 'Bertram', 'Chris', 'David'), ('A', 'B', 'C', 'D'), ('MH', 'SHH', 'MPSS7', 'LGH')]
>>> names, grades, highschools = tuple(list(zip(*alist)))
>>> names
('Albertina', 'Bertram', 'Chris', 'David')
>>> grades
('A', 'B', 'C', 'D')
>>> highschools
('MH', 'SHH', 'MPSS7', 'LGH')
```
And now for something completely different
Copying Data Structures

- Copying and assignment are two different things
Copying Data Structures

• Copying and assignment are two different things
  • We have an object \( a \)
    \[
    a = \text{set}(1, 2, \text{"one"})
    \]
  • We assign \( a \) to \( b \)
  • But the two objects are still linked:
Copying Data Structures

• Copying and assignment are two different things

```python
a = set([1, 2, "one"])
print(a)
b = a
print(b)
# Now we change set a
a.remove("one")
# Which also changes set b
print(b)
```

```python
>>> a = {1,2,"one"}
>>> a
{1, 2, 'one'}
>>> b = a
>>> a.remove("one")
>>> a
{1, 2}
>>> b
{1, 2}
```
Copying Data Structures

- Copying and assignment are two different things
- Here is what happens
  - In Python, names point to objects
  - Assigning adds a name to the same object

\[
\begin{array}{c}
\text{a} \rightarrow \{1, 2, \text{"one"}\} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a} \rightarrow \{1, 2, \text{"one"}\} \\
\text{b} & \\
\end{array}
\]
Copying Data Structures

• Copying and assignment are two different things
  • Since there is only one object, I can manipulate the object through either name
Copying Data Structures

• Copying and assignment are two different things
  • If I want to copy, I need to do so explicitly

```python
lista = [1, 2, "three", [4,5]]
listb = [x for x in lista]
lista[2] = 3
print(lista)
print(listb)
```

• Now changes to one do not change the other!

```python
>>> lista = [1, 2, "three", [4,5]]
>>> listb = [x for x in lista]
>>> lista[2] = 3
>>> lista
[1, 2, 3, [4, 5]]
>>> listb
[1, 2, 'three', [4, 5]]
```
Copying Data Structures

• Copying and assignment are two different things
  • One can use slices to copy lists
    • \[ \text{listb} = \text{lista}[0:4] \]
Copying Data Structures

• Copying becomes difficult if we have compound objects
  • E.g.: A list which contains lists, sets, …

• Shallow copy:
  • Resulting copies have shared elements
Copying Data Structures

• Example: A matrix as a list of rows
  • Create zero row by multiplying list with an integer

\[
\text{matrix} = 3 \times \left[ 4 \times [0] \right]
\]

• One might think it creates a structure like

\[
\left[ \left[ 0, 0, 0, 0 \right], \left[ 0, 0, 0, 0 \right], \left[ 0, 0, 0, 0 \right], \left[ 0, 0, 0, 0 \right] \right]
\]

• which is not entirely false
Copying Data Structures

• We can get the elements as we should

```python
matrix = 3*[ 4*[0]]
print(matrix[3][2])
```

• And we can set elements

```python
matrix = 3*[ 4*[0]]
matrix[3][2] = 5
```

• But now we see that we got three times the same row
Copying Data Structures

```python
matrix = 3*[ 4*[0] ]
print(matrix)
matrix[2][3] = 5
print(matrix)
```

```python
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
[[0, 0, 0, 5], [0, 0, 0, 5], [0, 0, 0, 5]]
```
Copying Data Structures

• How can we do this:
  • Need to construct the zero rows independently
    • Use e.g. list comprehension

```python
matrix = [ [0 for _ in range(4)] for i in range(3)]
```
Copying Data Structures

• Shallow copy: Assume we have
  \[
  \text{lista} = [1, 2, [3, 4, 5]]
  \]

• We create a shallow copy by
  \[
  \text{lista} = [1, 2, [3, 4, 5]]
  \]
  \[
  \text{listb} = \text{lista}[:]
  \]

• But here is what is happening
The two lists still share a component. We can change this component in one list and change it in the other one as well.
Copying Data Structures

- We have two copies of the list, but the third element are two different names for the same object

```python
lista = [1, 2, [3, 4, 5]]
listb = lista[:]
```
Copying Data Structures

- In consequence, I can alter the same element in the list which is element number 2

```python
lista = [1, 2, [3, 4, 5]]
listb = lista[:]
lista[2][0] = 6
print(lista)
print(listb)
```

- prints out

```
[1, 2, [6, 4, 5]]
[1, 2, [6, 4, 5]]
```
Copying Data Structures

• I need to use a deep copy
  • Easiest:
    • Use the module copy
      • Use `copy.deepcopy(object)` for deep copying
      • Use `copy.copy(object)` for shallow copying
Copying Data Structures

• This is a famous Python gotcha
  • Behavior is not intuitive.