# Programming Patterns in Python

Terence Parr
MSDS program
**University of San Francisco**

# Don Knuth on "geekhood"

*The main characteristic is an ability to understand many levels of abstraction simultaneously, and to shift effortlessly between in-the-large and in-the-small. A geek knows that, to achieve a certain high-level goal, you need to add one to a certain counter at a certain time.*

# How programmers design programs

- Experienced programmers draw from a collection of generic high-level / large-scale mental templates as starting points
- There are mental templates for desktop GUI apps, machine learning classifiers, web servers, etc....
- A template provides an overall structure for the program, like lawyers tweaking a contract from previous client for new one
- Engineers building a new suspension bridge do not proceed as if such a thing has never been built before
- Gaining experience as a programmer means recognizing and remembering patterns in your code

# Example: data science program template

1. Acquire data, which means finding a suitable file or collecting data from the web and storing in a file or database
2. Load data from disk or database and organize into a suitable data structure in memory
3. Normalize, filter, clean, or otherwise prepare data
4. Process the data, which can mean training a machine learning model, transforming the data, computing summary statistics, or optimizing a cost function
5. Emit results, which can be anything from simply printing an answer to saving data to the disk to generating a fancy visualization

# Low-level programming patterns

- High-level templates help us organize and plan our program
- Low-level patterns are puzzle pieces that we combine to fill in details and solve parts of the overall template
- These patterns have Python implementations but we design programs by selecting and applying patterns/operations, not specific code sequences
- When designing a program, I never say:
  "*Oh! I need a for-loop with an if-statement right here*."

# Sample programming patterns

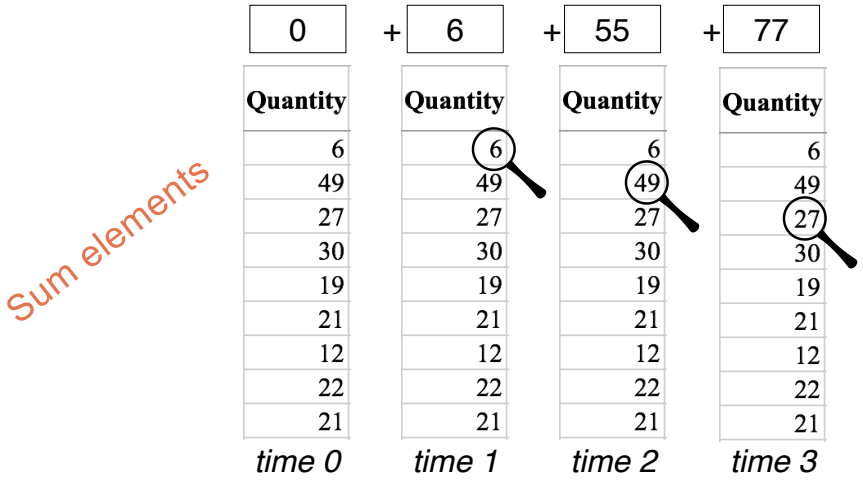
- You're no doubt familiar with simple patterns such as:
  - *sum the numbers in a list*
  - *count the elements in a list*
- But there are many many more, such as:
  - *find all values in a list satisfying a condition*
  - *apply an operation to each element of a list to get new list*
  - *merge two sorted lists*
  - *delete records in a dataframe that satisfy a condition*
- Think and plan at this level or higher, not the code level

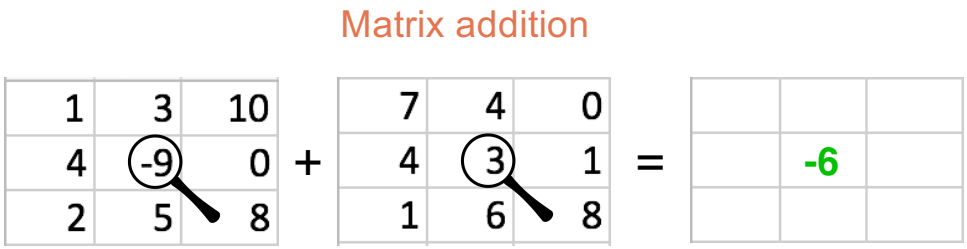
# A parade of patterns

UNIVERSITY OF SAN FRANCISCO

# Visualize behavior then identify pattern

- Think visually about how you would manipulate lists of data or extract information from data
- Manually moving some data around on paper or in spreadsheet helps me to understand the operation to perform



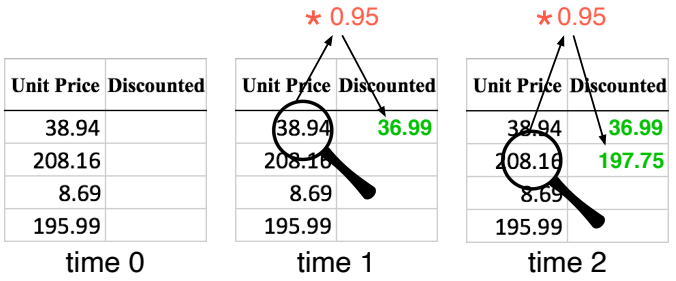
# Accumulate

- Traverses a sequence of elements and *accumulates* a value
- In Excel, this is like using **sum(...)** in a cell.
- Can use any other arithmetic operator, such as *
- Called *reduce*, as in *map*/*reduce* of distributed computing world

```
Quantity = [6, 49, 27, 30, 19, 21, 12, 22, 21]
sum = 0
for q in Quantity:
    sum += q          # same as: sum = sum + q
print(sum)
```

Experiment in pythontutor

# Map



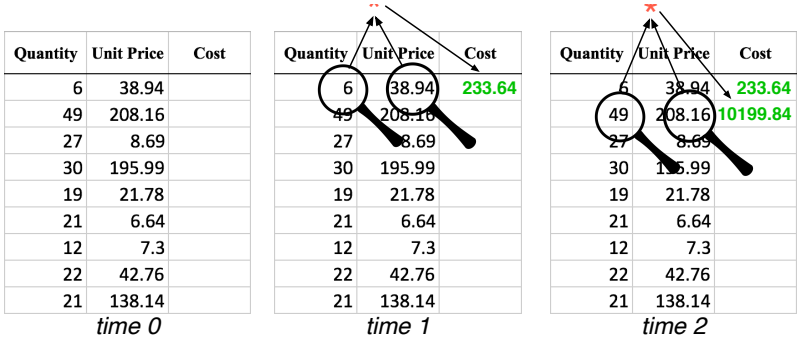

- A very common operation *maps* one sequence to another, applying an operator or function to each element
- It's like using a spreadsheet to create a new column containing some product unit price discounted by 5%

```
UnitPrice = [38.94, 208.16, 8.69, 195.99]

discounted = [] # empty list
for price in UnitPrice:
    discounted.append(price * 0.95)

# list comprehension shines here!!
discounted = [p*0.95 for p in UnitPrice]
```

Be able to reverse this, going from code to pattern!

Experiment in pythontutor

# Combine



| Quantity | Unit Price | Cost |
|---|---|---|
| 6 | 38.94 | |
| 49 | 208.16 | |
| 27 | 8.69 | |
| 30 | 195.99 | |
| 19 | 21.78 | |
| 21 | 6.64 | |
| 12 | 7.3 | |
| 22 | 42.76 | |
| 21 | 138.14 | |

*time 0*

| Quantity | Unit Price | Cost |
|---|---|---|
| 6 | 38.94 | 233.64 |
| 49 | 208.16 | |
| 27 | 8.69 | |
| 30 | 195.99 | |
| 19 | 21.78 | |
| 21 | 6.64 | |
| 12 | 7.3 | |
| 22 | 42.76 | |
| 21 | 138.14 | |

*time 1*

| Quantity | Unit Price | Cost |
|---|---|---|
| 6 | 38.94 | 233.64 |
| 49 | 208.16 | 10199.84 |
| 27 | 8.69 | |
| 30 | 195.99 | |
| 19 | 21.78 | |
| 21 | 6.64 | |
| 12 | 7.3 | |
| 22 | 42.76 | |
| 21 | 138.14 | |

*time 2*

- Traverse two lists at once placing the result in a third list
- Multiply the $i^{th}$ element from two different sequences and placing the result in the $i^{th}$ position of the output sequence

```
Quantity = [6, 49, 27, 30]
UnitPrice = [38.94, 208.16, 8.69, 195.99]

cost = []
for i in range(len(Quantity)):
    cost.append( Quantity[i] * UnitPrice[i] )

Cost = [Quantity[i] * UnitPrice[i] for i in range(len(Quantity))]
```

Experiment in pythontutor

# Split

- The opposite of combining is splitting where we split a stream into two or more new streams
- Example: split list of full names into their first and last names

```
names = ['Terence Parr', 'Diane Woodbridge', 'Yannet Interian']
for name in names:
    print(name.split())

first = []
last = []
for name in names:
    f,l = name.split()
    first.append(f)
    last.append(l)
```

| E | F |
|---|---|
| **Customer Name** | |
| Muhammed MacIntyre | |
| Barry French | |
| Barry French | |
| Clay Rozendal | |
| Carlos Soltero | |
| Carlos Soltero | |

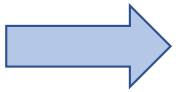
| E | F |
|---|---|
| **First Name** | **Last Name** |
| Muhammed | MacIntyre |
| Barry | French |
| Barry | French |
| Clay | Rozendal |
| Carlos | Soltero |
| Carlos | Soltero |

[Experiment in pythontutor]

# Slice a list (or string)

- Some operations yield subsets of the data, such as slice, which extracts a subset of a list (that fits in memory)
- Syntax is **A[begin:end]** where **begin** is inclusive and **end** is not

```
names=['Xue', 'Mary', 'Bob']
print(names[0:1])
print(names[0:2])
print(names[0:3])
print(names[2:3])
print(names[1:])
print(names[-1])
print(names[-2:])
```



```
['Xue']
['Xue', 'Mary']
['Xue', 'Mary', 'Bob']
['Bob’]
['Mary', 'Bob']
Bob
['Mary', 'Bob']
```

Experiment in pythontutor

# Filter

- The filter operation is similar to the map operation in that a computation is applied to each element of the input stream
- Filter tests each element for a specific condition and, if true, adds that element to the new sequence

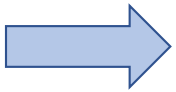
```
shipping = [35, 68.02, 2.99, 3.99, 5.94, 4.95, 7.72, 6.22]
shipping2 = []
for x in shipping:
    if x < 10:
        shipping2.append(x)
print(shipping2) # prints [2.99, 3.99, 5.94, 4.95, 7.72, 6.22]

shipping2 = [x for x in shipping if x < 10] # much easier
```

Experiment in pythontutor

# Filtering rows of data

- We can also filter on one column but keep the data within each row together; e.g., filter for Oscar winners

```
oscars = [
    [1984, "A Soldier's Story", 0],
    [1984, 'Places in the Heart', 0],
    [1984, 'The Killing Fields', 0],
    [1984, 'A Passage to India', 0],
    [1984, 'Amadeus', 1],
    [1985, "Prizzi's Honor", 0],
    [1985, 'Kiss of the Spider Woman', 0],
    [1985, 'Witness', 0],
    [1985, 'The Color Purple', 0],
    [1985, 'Out of Africa', 1]
]
print([movie for movie in oscars if movie[2]==1])
```



```
[[1984, 'Amadeus', 1], [1985, 'Out of Africa', 1]]
```

Experiment in pythontutor

# Search

- The filter operation finds all elements in a sequence that satisfy a specific condition, but often we'd like to know which element satisfies the condition first (or last)
- Search returns the first (or last) position in the sequence rather than the value at that position

```
first=['Xue', 'Mary', 'Robert']     # our given input
target = 'Mary'                     # searching for Mary
index = -1
for i in range(len(first)):         # i is in range [0..n-1] or [0..n)
    if first[i]==target:
        index = i
        break
print(index)
```

Experiment in pythontutor

# Grid/matrix processing

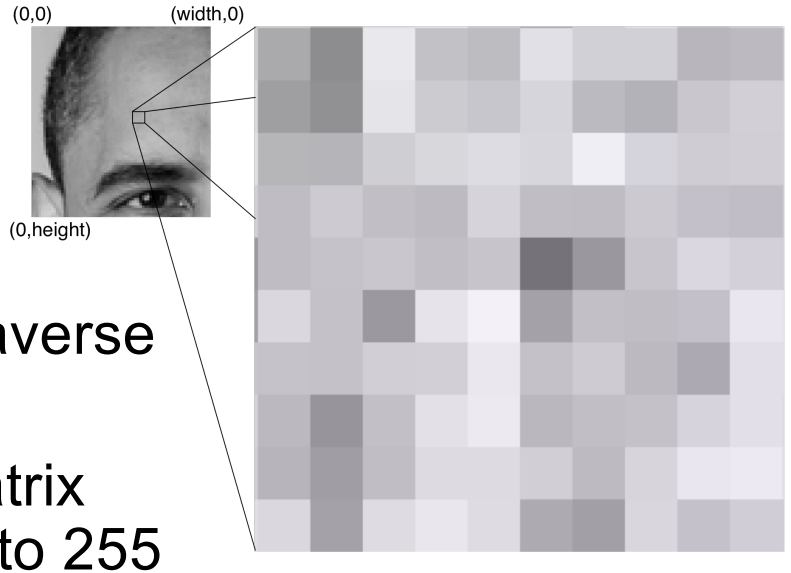
| | | |
|---|---|---|
| 1 | 3 | 10 |
| 4 | -9 | 0 |
| 2 | 5 | 8 |

- Any time you need to process each cell in a two dimensional structure such as an image or matrix, think "nested loop"
- Single loop does 1D, nested loop does 2D, triple loop does 3D...

```
nrows = 3
ncols = 3
# column j value varies more quickly than the row i value
for i in range(nrows):
    for j in range(ncols):
        print( i, j )

# row i value varies more quickly than j
for j in range(ncols):
    for i in range(nrows):
        print( i, j )
```

Experiment in pythontutor

# Image processing




- The images project requires that you traverse the x,y coordinates of images
- An image is nothing more than a 2D matrix whose entries are grayscale pixels in 0 to 255
- A pixel value of 0 is black and 255 is white

```
# walk top-down: row-by-row of pixels
for y in range(height):
    for x in range(width):
        print( x, y )
```

# Applying patterns

Now that we can read basic Python and have seen some pattern implementations, let's use those patterns to solve some simple problems

# Basic problem-solving strategy

- **Start with the end result and work your way backwards**
- Ask what the prerequisites are for each step
- The processing step or steps preceding step *i* compute the data or values needed by step *i*
- E.g., median: to pick middle value, previous step must sort data
- Data science problems are often solved with an "iterative refinement of data" approach to arrive at a final result

# Exercise

- Given a list of prices, cut the price of each in half

```
prices = [38.94, 208.16, 8.69, 195.99]
```

- First, must decide if we are altering in place or creating a new; let's create a new list in general for these exercises
- Which pattern should we apply?  Map...what does code look like?

```
prices = [p/2 for p in prices]
```

Experiment in pythontutor

# Exercise

- Given a list of names, get a list of string lengths called **namelens**; function **len(x)** yields length of lists, strings, etc...

```
names = ['Xue', 'Mary', 'Robert']
```

- Draw out what the transformation looks like

```
['Xue', 'Mary', 'Robert'] ☞ [3, 4, 6]
```

- Which pattern should we apply? Map...what does code look like?

```
namelens = [len(name) for name in names]
```

Experiment in pythontutor

# Exercise

- Given a list of prices, get a list of prices greater than $100

```
prices = [38.94, 208.16, 8.69, 195.99]
```

- Which pattern should we apply? Filter...what does code look like?

```
prices = [p for p in prices if p>100]
```

Experiment in pythontutor

# Exercise

- Given a list of prices, double any price less than $10

```
prices = [38.94, 208.16, 8.69, 195.99]
```

- Which patterns should we apply? Filter & Map

```
prices = [p for p in prices if p<10]
prices = [p/2 for p in prices]

prices = [p/2 for p in prices if p<10] # Or together
```

Experiment in pythontutor

# Exercise

- Given a list of movie titles, how many are 3-words long?

```
titles = ["A Soldier's Story", 'Places in the Heart', 'The Killing Fields',
          'A Passage to India', 'Amadeus', "Prizzi's Honor",
          'Kiss of the Spider Woman', 'Witness', 'The Color Purple',
          'Out of Africa']
```

- Which patterns should we apply and how?
- What does the problem-solving process look like?

Think how we can gradually morph the data from titles to result, but start with the desired result and work backwards

# Exercise continued

```
["A Soldier's Story", 'Places in the Heart',
 'The Killing Fields', 'A Passage to India',
 'Amadeus', "Prizzi's Honor", 'Kiss of the Spider Woman',
 'Witness', 'The Color Purple', 'Out of Africa']
```

- Work backwards:
  - If we have the list of title lengths, we can filter for length 3 and count

```
[3, 4, 3, 4, 1, 2, 5, 1, 3, 3]  ☞  [3, 3, 3, 3]  ☞  4
```

  - To get list of title lengths, map **len()** to list of words of each title

```
[['A', "Soldier's", 'Story'],              ☞  [3, 4, ...]
 ['Places', 'in', 'the', 'Heart'], ...]
```

  - To get the list of words for each title, map **split()** to each title

```
["A Soldier's Story",
 'Places in the Heart',      ☞   [['A', "Soldier's", 'Story'],
 ...]                             ['Places', 'in', 'the', 'Heart'],
                                  ...]
```

# Exercise continued

```
["A Soldier's Story", 'Places in the Heart',
 'The Killing Fields', 'A Passage to India',
 'Amadeus', "Prizzi's Honor", 'Kiss of the Spider Woman',
 'Witness', 'The Color Purple', 'Out of Africa']
```

- Now, reverse it to get the correct sequence:

**Map** `["A Soldier's Story", 'Places in the Heart', ...]` ☞ `[['A', "Soldier's", 'Story'], ['Places', 'in', 'the', 'Heart'], ...]`

**Map** `[['A', "Soldier's", 'Story'], ['Places', 'in', 'the', 'Heart'], ...]` ☞ `[3, 4, ...]`

**Filter** `[3, 4, 3, 4, 1, 2, 5, 1, 3, 3]` ☞ `[3, 3, 3, 3]` ☞ `4`

# Exercise continued

```
titles = ["A Soldier's Story", 'Places in the Heart', 'The Killing Fields',
          'A Passage to India', 'Amadeus', "Prizzi's Honor",
          'Kiss of the Spider Woman', 'Witness', 'The Color Purple',
          'Out of Africa']

tsplit = [t.split() for t in titles] # Map
tlen = [len(t) for t in tsplit]      # Map
t3 = [n for n in tlen if n==3]       # Filter
n = len(t3)                          # Prints 4

t3 = [t for t in titles if len(t.split())==3] # Map/Filter
n = len(t3)
```

Experiment in pythontutor