Python Data Collection and Management for Public Policy Research

Day 6: Basic Python (Part 3)

Blake Miller[†]

8 July, 2024

[†]Assistant Professor, Department of Methodology, London School of Economics and Political Science (E-mail: b.a.miller@lse.ac.uk)

Agenda for Today

- Basic Python
  - Regex in Python
  - Recursion
  - Lists and Tuples
  - Dictionaries
  - Files
  - Exceptions and Exception Handling
- Coding Session: Applying Iteration and Conditional Statements

Searching with `re.search`

- `re.search(pattern, string)` searches the string for the first location where the regex pattern produces a match.
- Returns a match object if a match is found, otherwise 'None'.

```
>>> import re
>>> search = re.search(r'[A-Za-z]+\-[A-Za-z
    ]+', 'LSE-Fudan Summer School')
>>> print(search)
<re.Match object; span=(0, 9), match='LSE-
    Fudan'>
>>> print(search.start())
0
>>> print(search.end())
9
>>> print(search.group())
LSE-Fudan
```

Using `re.sub` for Substitutions

`re.sub(pattern, repl, string)` replaces occurrences of the 'pattern' in 'string' with 'repl'

```
>>> import re
>>> result = re.sub(r'[A-Za-z]+\-[A-Za-z]+',
    'Day 5 of', 'LSE-Fudan Summer School')
>>> print(result)
Day 5 of Summer School
>>> result = re.sub(r'[A-Za-z]+\-[A-Za-z]+',
    'Day 5 of', 'LSE-Fudan Summer School')
>>> print(result)
Day 5 of Summer School
>>> result = re.sub(r'([A-Z]+|[0-9]+)', '***
   ', 'ABC abc 123')
>>> print(result)
*** abc ***
```

Reading Word Lists

- One common task is reading through lists of words for analysis.
- Python's file handling makes it easy to read and process text files.
- Example of reading from a file:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

Building on Regular Expressions

- Regular Expressions (Regex) Tutorial by Corey Schafer on YouTube
- Interactive Regex Tutorial at RegexOne
- Applications of Regex to Text Processing by Monica Pérez Nogueras

Coding Session: Reading in and Standardizing User Inputted Data

Recursion

Recursion

A function can call itself to loop through data to reach a result.

```
>>> def countdown(n):
...     if n <= 0:
...         print('Blastoff!')
...     else:
...         print(n)
...         countdown(n-1)
...
>>> countdown(3)
3
2
1
Blastoff!
```

Recursion Example

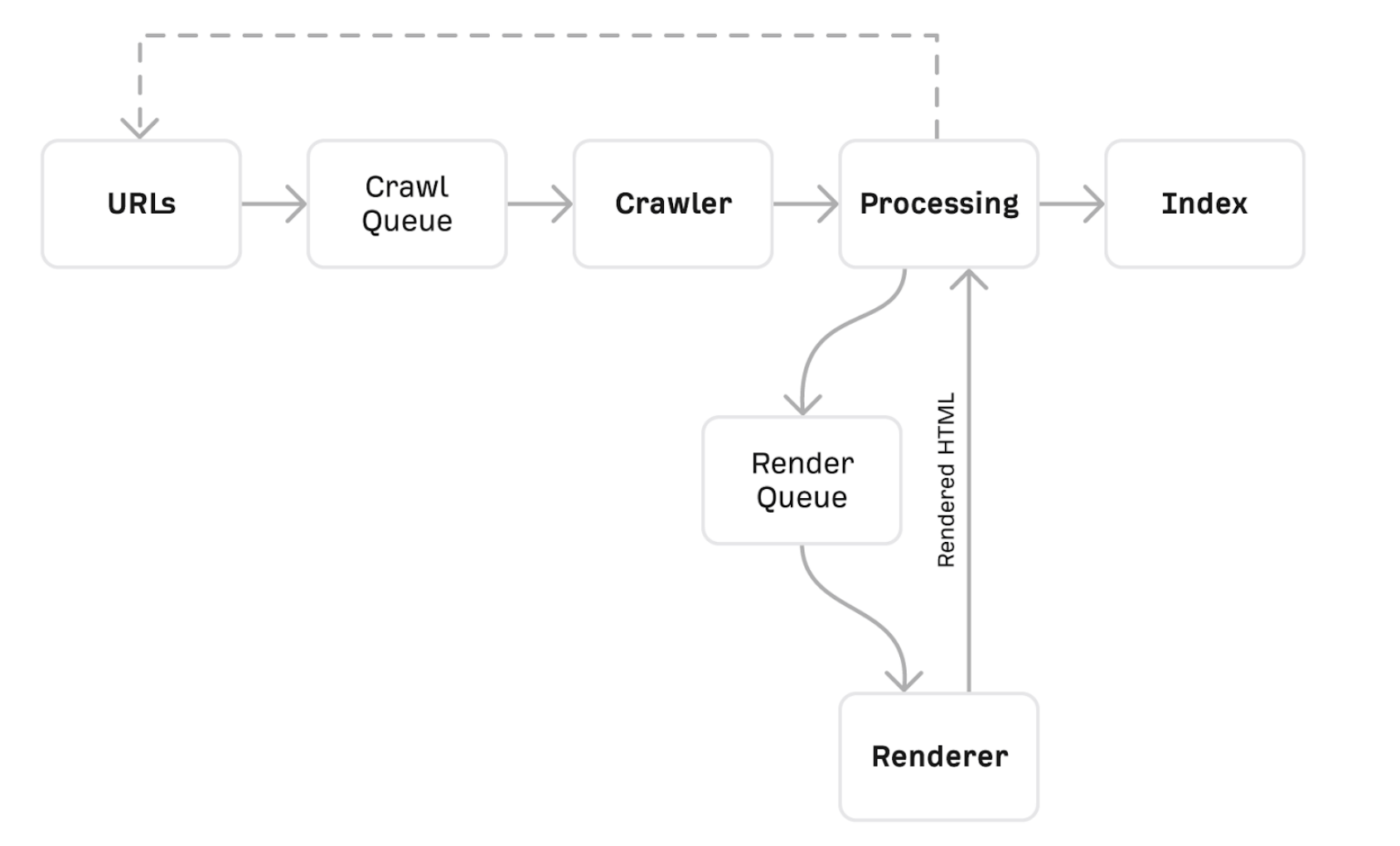
```
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> print(factorial(5))
120
```

Checking Types

Python allows you to check and enforce the types of arguments that a function can accept.

```
>>> def factorial(n):
...     if not isinstance(n, int) or n < 0:
...         print("Invalid input.")
...         return None
...     elif n == 0:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> print(factorial("fred"))
Invalid input.
None
```

Recursion in the Real World: Crawlers



Source: ahrefs Blog

Lists

A List is a Sequence

- Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type.
- Lists are mutable, which means we can change their elements.

```
>>> profs = ['Meng', 'Hildebrandt', 'Ding',
    'Miller', 'Puppim de Oliveira', 'Mendez',
     'Alden']
>>> print(profs)
['Meng', 'Hildebrandt', 'Ding', 'Miller', '
    Puppim de Oliveira', 'Mendez', 'Alden']
```

Lists are Mutable

The fact that lists are mutable means you can modify the elements of a list in place.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

Traversing a List

The most common way to traverse the elements of a list is with a simple for loop.

```
>>> for prof in profs:
...     print("Dr.", prof)
...
Dr. Meng
Dr. Hildebrandt
Dr. Ding
Dr. Miller
Dr. Puppim de Oliveira
Dr. Mendez
Dr. Alden
```

List Operations

The ‘+‘ operator concatenates lists.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

List Slices

Recall string slicing from day 4 lecture. Slicing works on lists just as with strings.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

List Methods

Lists have various methods, such as ‘append‘, ‘extend‘, ‘sort‘, etc.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
>>> t.extend(['e', 'g', 'f'])
>>> print(t)
['a', 'b', 'c', 'd', 'e', 'g', 'f']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Deleting Elements

Elements can be removed from a list using `pop`, `remove`, `del`.

- Using `pop` to remove an element at a specified index and return it:

```
>>> t = ['a', 'b', 'c', 'd']
>>> x = t.pop(1)
>>> print(t)
['a', 'c', 'd']
>>> print(x)
b
```

Deleting Elements

Elements can be removed from a list using `pop`, `remove`, `del`.

- Using `remove` to delete the first occurrence of a value:

```
>>> t = ['a', 'b', 'c', 'b']
>>> t.remove('b')
>>> print(t)
['a', 'c', 'b']
```

Deleting Elements

Elements can be removed from a list using `pop`, `remove`, `del`.

- Using `del` to remove an element by index:

```
>>> t = ['a', 'b', 'c', 'd']
>>> del t[1]
>>> print(t)
['a', 'c', 'd']
```

Dictionaries

A Dictionary is a Mapping

A dictionary maps keys to values and provides a fast way to access the data.

```
>>> class_teachers = {
...     'Hildebrandt': 'Social Governance
    and Policy Innovation',
...     'Meng': 'Chinese Media, Global
    Contexts',
...     'Miller': 'Python Data Collection
    and Management'
... }
>>> print(class_teachers['Miller'])
Python Data Collection and Management
```

Dictionary Operations

Dictionaries support various operations like adding, modifying, and removing entries.

```
>>> class_teachers['Ding'] = 'Comparative
    Public Policy'  # Adds a new entry
>>> class_teachers['Miller'] = 'Python for
    Public Policy Research'  # Modifies an
    existing entry
>>> print(class_teachers)
{'Hildebrandt': 'Social Governance and
    Policy Innovation', 'Meng': 'Chinese
    Media, Global Contexts', 'Miller': '
    Python for Public Policy Research', 'Ding
    ': 'Comparative Public Policy'}
```

Dictionary as a Set of Counters

You can use dictionaries to count occurrences of items.

```
>>> # counting the letters in a word
>>> def histogram(s):
...         d = {}
...         for c in s:
...             if c not in d:
...                 d[c] = 1
...             else:
...                 d[c] = d[c] + 1
...         return d
...
>>> h = histogram('brontosaurus')
>>> print(h)
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's'
    : 2, 'a': 1, 'u': 2}
```

Looping and Dictionaries

You can loop through the keys in a dictionary using a ‘for‘ loop.

```
>>> student_count = {'Miller': 10, '
    Hildebrandt': 15, 'Ding': 10}
>>> for prof in student_count:
...     print(prof, student_count[prof])
...
Miller 10
Hildebrandt 15
Ding 10
```

Reverse Lookup

Finding a key given a value is called a reverse lookup.

```
>>> def reverse_lookup(d, v):
...     for k in d:
...         if d[k] == v:
...             return k
...     raise ValueError('Value does not
    appear in the dictionary')
...
>>> reverse_lookup(h, 2)
'r'
```

Dictionaries and Lists

Lists can be used as values in a dictionary.

```
>>> d = {'a': [1, 2, 3], 'b': [4, 5, 6]}
>>> d['a'].append(4)
>>> print(d)
{'a': [1, 2, 3, 4], 'b': [4, 5, 6]}
```

Tuples

Tuples are Immutable

- A tuple is an immutable sequence of values.
- Once created, the values in a tuple cannot be changed.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
a
>>> t[0] = 'A'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
    item assignment
```

Tuple Assignment

Tuple assignment allows multiple variables to be assigned at once.

```
a, b = 1, 2
print(a)
print(b)
```

Tuples as Return Values

Functions can return tuples; useful for returning multiple values.

```
>>> def minutes_to_hours(minutes):
...     hours = minutes // 60
...     remainder = minutes - hours * 60
...     print(hours, "hour", remainder, "
    minutes")
...     return(hours, remainder)
...
>>> hours, minutes = minutes_to_hours(185)
3 hour 5 minutes
>>> hours
3
>>> minutes
5
```

Variable-length Argument Tuples

Functions can take a variable number of arguments using `*args`.

```
>>> def printall(*args):
...         print(args)
...
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Lists and Tuples

Lists and tuples can be used together to perform various operations using `zip` to create a list of tuples:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zipped = list(zip(s, t))
>>> print(zipped)
[('a', 0), ('b', 1), ('c', 2)]
```

Iterating through of Lists of Tuples

You can iterate through lists of tuples as follows:

```
>>> t = [('a', 0), ('b', 1), ('c', 2)]
>>> for letter, number in t:
...     print(letter, number)
...
a 0
b 1
c 2
```

List and Dictionary Comprehensions

List Comprehensions

Shortcut to create lists from sequences or ranges.

```
>>> numbers = [1, 2, 3, 4, 5]
>>> squares = [x**2 for x in numbers]
>>> print(squares)
[1, 4, 9, 16, 25]
```

More List Comprehensions

Can aid in performing calculations on each element in a list.

```
>>> names = ['Yuki', 'Jorge', 'Mei', 'Aya']
>>> lengths = [len(name) for name in names]
>>> print(lengths)
[4, 5, 3, 3]
```

Dictionary Comprehensions

Dictionary comprehensions are similar in syntax.

```
>>> names = ['Yuki', 'Jorge', 'Mei', 'Aya']
>>> name_lengths = {name: len(name) for name
    in names}
>>> print(name_lengths)
{'Yuki': 4, 'Jorge': 5, 'Mei': 3, 'Aya': 3}
```

Using Conditionals in List Comprehensions

We can use conditions within a list comprehension to filter items.

```
>>> ages = [22, 35, 27, 21, 40]
>>> adults = [age for age in ages if age >=
    21]
>>> print(adults)
[22, 35, 27, 21, 40]
```

Dictionary Comprehensions with Conditionals

Dictionary comprehension also support conditional logic.

```
>>> ages = {'Yuki': 22, 'Jorge': 17, 'Mei':
    25, 'Aya': 21}
>>> adults_dict = {name: age for name, age
    in ages.items() if age >= 21}
>>> print(adults_dict)
{'Yuki': 22, 'Mei': 25, 'Aya': 21}
```

Nested List Comprehensions

Like for loops, list comprehensions can be nested. Here we flatten a matrix (list of lists)

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8,
    9]]
>>> flattened = [num for row in matrix for
    num in row]
>>> print(flattened)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Building on Lists, Dictionaries

- Instructional video on list and dictionary comprehensions (watch here)
- Tutorial on using the Counter and collections module in Python (read here)