

Python Data Collection and Management for Public Policy Research

Day 5: Basic Python (Part 2)

Blake Miller[†]

5 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
 - Conditional Statements
 - Iteration
 - Strings
 - Regular Expressions
 - Recursion
- Coding Session: Applying Iteration and Conditional Statements

Conditional Statements

Comparison Operators for int, float, string

Comparison operators are used to compare two values. The outcome of these comparisons is a Boolean value (True or False).

Operator	Description
==	Equality
!=	Inequality
is	Identity (True)
is not	Identity (False)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Equality vs. Assignment Operators

- The **assignment Operator** (=) is used to assign the value on the right to the variable name on the left.
- The **equality Operator** (==) is used to compare two values to check if they are equal.

```
>>> x = 10 # Assigns value of 10 to 'x'
>>> x == 10 # Checks equality of 'x' and 10
True
>>> x == 5 # Checks equality of 'x' and 5
False
>>> x = 5 # Assigns value of 5 to 'x'
>>> x == 5 # Checks equality of 'x' and 5,
          again
True
```

Boolean Expressions

A Boolean expression is an expression that is either True or False.

```
>>> 1 > 2
False
>>> "apple" != "banana"
True
>>> 5 == 5
True
>>> 10 <= 10
True
```

Logic Operators

Logic operators are used to combine Boolean values.

```
>>> a = True
>>> b = False
>>> a and b
False
>>> a or b
True
>>> a and not b
True
>>> b and not a
False
>>> a or not b
True
>>> b or not a
False
```

Equality vs. Identity

- Identity operators usually work the same as equality operators, but they are slightly different.
- They check if two variables point to the same object in memory, not just if they are equal.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = a
>>> a == b  # Same value, different object
True
>>> a is b   # Not the same object
False
>>> a is c   # 'c' and 'a' refer to the same
              object
True
```


Combining Boolean Expressions and Logical Operators

```
>>> x = 15
>>> y = 10
>>> print(x > 10 and y < 15)
True
>>> if x > 10 and y < 15:
...     print("Both conditions met.")
... else:
...     print("Conditions not met.")
...
Both conditions met.
```

if Statements

- Conditional statements give the ability to check conditions and change the behavior of the program.
- if statements:

```
>>> x = 42
>>> if x > 0:
...     print('x is positive')
...
x is positive
```

else Statements

The else statement offers an alternative action.

```
>>> x = -42
>>> if x > 0:
...     print('x is positive')
... else:
...     print('x is not positive')
...
x is not positive
```

elif Statements

elif allows multiple conditions to be chained.

```
>>> x = 0
>>> if x > 0:
...     print('x is positive')
... elif x < 0:
...     print('x is negative')
... else:
...     print('x is zero')
...
x is zero
```

Nested Conditionals

One conditional can also be nested within another.

```
>>> x = 42
>>> y = 3
>>> if x == 42:
...     if y == 3:
...         print('x is 42 and y is 3')
...
x is 42 and y is 3
```

Iteration

range() Function

- The range() function creates a range of integers between two values: range(start, stop, step)
- start and step are optional, with values start = 0, step = 1
- Range of values is between start and stop - 1

```
>>> for i in range(5, 15, 5):  
...     print(i)  
...  
5  
10
```

for Loops

for loops iterate over any set of values.

```
>>> for i in range(10):  
...     if i % 2 == 0:  
...         print(i)  
...  
0  
2  
4  
6  
8
```


while Loops

A while loop continues executing as long as the loop condition remains true.

```
>>> i = 5
>>> while i > 0:
...     print(i)
...     n -= 1
...
5
4
3
2
1
>>> print('Blastoff!')
Blastoff!
```

The break Statement

- The break statement is used to exit a loop prematurely.
- It ends the nearest enclosing loop, skipping any remaining code in the loop.

```
>>> count = 0
>>> for i in range(100):
...     count += 1
...     if i == 10:
...         break
...
>>> print(count)
10
```

The continue Statement

- The continue statement is used to skip to the next iteration of a loop prematurely.

```
>>> count = 0
>>> for i in range(10):
...     if i % 2 == 0:
...         continue
...     count += 1
...
>>> print(count)
5
```

Working with Strings

A String is a Sequence

- A string is a sequence of case-sensitive characters.
- You can access characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print(letter)
a
```

- Square brackets are used to perform indexing into a string to get the value at a certain index/position.
- Indexing always starts at 0. The last element is always at index -1.

String Indexing Example

```
>>> s = "abc"
>>> s[0]
'a'
>>> s[1]
'b'
>>> s[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[-1]
'c'
>>> s[-2]
'b'
```

- A string slice is a segment of a string. Selecting a slice is similar to selecting a character:
 - Can slice strings using `[start:stop:step]`
 - If give two numbers, `[start:stop]`, `step=1` by default
 - You can also omit numbers and leave just colons

String Slicing Example

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
>>> s[:2]
'Mn'
>>> s[6:12:3]
'Ph'
>>> s[::-1]
'nohtyP ytnoM'
```

Strings are Immutable

- Strings are “immutable” and cannot be modified.
- Once a string is created, the characters within it cannot be changed.

```
>>> s = "hello"
>>> s[0] = 'y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
    item assignment
>>> s = 'y' + s[1:len(s)]
```

- The above operation binds `s` to a new object, not modifying the original string.

Length of Strings: len()

The 'len()' function returns the number of characters in a string.

```
>>> fruit = 'banana'
>>> length = len(fruit)
>>> print(length)
6
```

Traversal with a for Loop

You can loop through the characters in a string with a for loop:

```
>>> fruit = 'banana'
>>> for char in fruit:
...     print(char)
...
b
a
n
a
n
a
```

String Methods and Operators

- Strings in Python come equipped with a variety of built-in methods.
- A **method** is similar to a function in that it takes arguments and returns a value. However, the syntax and context of use differ.
- Methods are always called on an object and operate within the context of that object.
- For example, string methods modify or interact with the string instance they are called on.

String Methods: Capitalization

```
>>> program = 'lse-fudan summer school'
>>> program.lower()
'lse-fudan summer school'
>>> program.upper()
'LSE-FUDAN SUMMER SCHOOL'
>>> program.title()
'Lse-Fudan Summer School'
```

String Methods: Stripping Whitespace

```
>>> program = ' lse-fudan summer school '  
>>> program.strip()  
'lse-fudan summer school'  
>>> program.rstrip()  
' lse-fudan summer school '  
>>> program.lstrip()  
'lse-fudan summer school '
```

String Methods: Finding and Replacing

```
>>> program = 'lse-fudan summer school'
>>> program.startswith('lse')
True
>>> program.endswith('ool')
True
>>> program.count('summer')
1
>>> program.replace('summer school', '
    conference')
'lse-fudan conference'
>>> program.find('fudan')
4
```


The in Operator

The `in` operator returns `True` if a character or substring exists within the string, otherwise `False`.

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
>>>
```


String Comparison

Strings can be compared with operators like “==,” “>,” “<,” etc.

```
>>> word = 'banana'
>>> if word == 'banana':
...     print('All right, bananas.')
...
All right, bananas.
```

Regular Expressions

What are Regular Expressions?

- Regular expressions (regex) are patterns used to match character combinations in text.
- They allow us some flexibility in search so we can find certain kinds of data we might want to extract or quantify.
- Commonly used in data cleaning, data extraction, and complex data analysis.
- Great interactive tutorial on [RegexOne](#) 

Why Learn Regular Expressions?

- Quickly find and replace patterns in text and data.
- Perform complex text matching and extraction, even when data are non-standard or dirty.
- When collecting data from online, it helps in the parsing of raw data.

- Finding specific patterns within data sets, like dates, emails, and phone numbers.
- Cleaning and preparing data for analysis.
- Automating the extraction of structured data from text, such as extracting all hyperlinks from a webpage.

Key Terms in Regular Expressions

Metacharacters Special characters that control the logic of a pattern in regular expressions.

Pattern The format or sequence that a regular expression defines, which is used to match against strings.

Character Class A set of characters enclosed within square brackets `[]` that matches any single character within the brackets. For example, `[abc]` matches "a", "b", or "c".

Key Terms in Regular Expressions

Grouping Parentheses () are used to group parts of expressions so that quantifiers or other operations can be applied to the entire group.

Greedy Match The default behavior of quantifiers that capture as much of the string as possible.

Lazy Match Quantifiers followed by a ? that modify them to capture as little of the string as possible, such as *? or +?.

Anchors Special metacharacters ^ and \$ that do not match characters but rather the positions before or after characters. Used to match a position before, after, or between characters.

Escape Characters The backslash \ is used to escape metacharacters so that they are treated as ordinary characters.

Basic Metacharacters

Character	Description
.	Matches any single character except newline.
*	Matches zero or more of the preceding element.
+	Matches one or more of the preceding element.

Special Character Classes

Character	Description
<code>\d</code>	Matches any digit (equivalent to <code>[0-9]</code>).
<code>\w</code>	Matches any word character (alphanumeric & underscore).
<code>\s</code>	Matches any whitespace character (spaces, tabs, line breaks).

Using Brackets and Hyphens

Character	Description
[abc]	Matches any of 'a', 'b', or 'c'.
[a-z]	Matches any lowercase letter from 'a' to 'z'.
[A-Za-z]	Matches any letter regardless of case.

Grouping in Regular Expressions

Character	Description
()	Groups parts of the expression. Useful for: <ul style="list-style-type: none">• Applying quantifiers to sequences as a single unit.• Capturing substrings for back-referencing.• Using alternation within the group (e.g., (dog cat)).
	Represents alternation (logical OR), used within groups to match one of several patterns.

Lookahead and Lookbehind in Regular Expressions

Feature	Description
(?=...)	Positive Lookahead: Asserts that what immediately follows the current position in the string is the pattern specified inside the parentheses, without including it in the match.
(?!...)	Negative Lookahead: Asserts that what immediately follows the current position in the string is not the pattern specified inside the parentheses.
(?<=...)	Positive Lookbehind: Asserts that what immediately precedes the current position in the string is the pattern specified inside the parentheses, without including it in the match.
(?<!...)	Negative Lookbehind: Asserts that what immediately precedes the current position in the string is not the pattern specified inside the parentheses.

Regular Expressions for Non-Latin Characters

You can use regular expressions with non-Latin characters using Unicode ranges. Unicode is the standard text input system used by most computers as a default.

Expression	Description
<code>[\u4e00-\u9fff]</code>	Matches any character in the range of common Chinese characters (Unicode range for CJK Unified Ideographs).
<code>[\u3400-\u4DBF]</code>	Matches characters in the Unicode range for CJK Unified Ideographs Extension A, less commonly used but still valid Chinese characters.
<code>[\u20000-\u2A6DF]</code>	Matches characters in the range of CJK Unified Ideographs Extension B, which includes historical and rare characters.
<code>[\u2A700-\u2B73F]</code>	Matches characters in the range of CJK Unified Ideographs Extension C.

Coding Session: Searching in Sublime Text with Regular Expressions

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 Noguerras

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

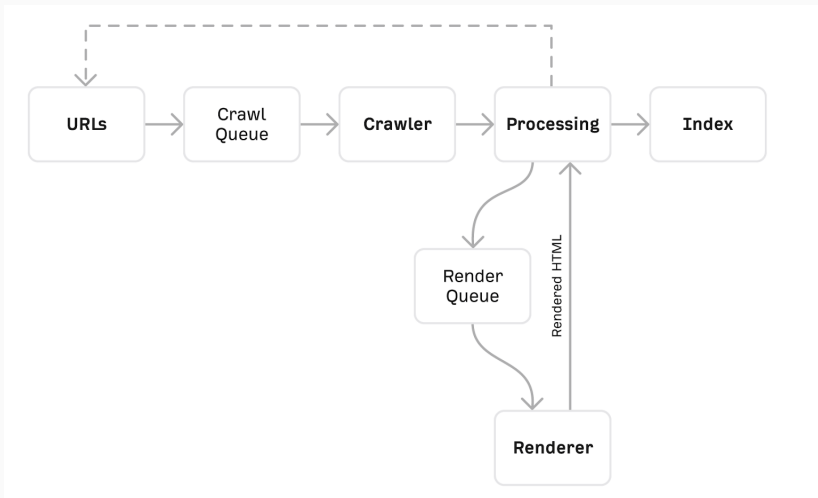
```
>>> def factorial(n):  
...     if n == 0:  
...         return 1  
...     else:  
...         return n * factorial(n-1)  
...  
>>> print(factorial(5))  
120
```

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](#) 