Introduction to Computation for the Humanities and Social Sciences

CS 3
Chris Tanner
Lecture 4

Python: Variables, Operators, and Casting
“[People] need to learn code, man I’m sick with the Python.”

— Childish Gambino
HOT 97 “freestyle”, Sept 8, 2014
A REFRESHER — IDE (Atom, Sublime, etc)

• Atom, Sublime, etc are **not** Python-specific

• They are programs just like any other on our computer

• They are text editors (like Microsoft Word, Text Edit, Notepad), which allow us to display and edit text

• However, they’re catered for the task of **programming**, so they have nifty features like displaying certain words w/ colors to make it easy to read

• Related, we could write Python code in Microsoft Word if we wanted, just that would make our life more difficult
CHICAGO — Gun violence this past weekend in Chicago injured 54 people and claimed 12 lives. Sadly, bloody weekends are practically a cyclical occurrence in Chicago during late summer and early fall.

In August 2017, 63 people were shot and eight killed in one weekend. In 2016, 59 were shot and 17 killed the weekend before Halloween. In July 2015, 40 were shot and eight were killed in one weekend. Accompanying that pattern of violence is the spurt of national outrage and partisan criticism that follows.

“Chicago murders are direct result of one party Democratic rule for decades,” tweeted Rudy Giuliani, President Trump’s lead lawyer and a former mayor of New York, in response to the tragedy. “Policing genius Jerry McCarthy can do for Chicago what I did for NYC. He was one of the architects of Compstat. Slashed homicides over 70%. Tens of thousands of lives saved.”

Mr. Giuliani was referring to Garry McCarthy, a former New York City police commander, former Chicago police superintendent and Mayor Rahm Emanuel’s electoral opponent. Mr. McCarthy disavowed the president’s and Mr. Giuliani’s politics while journalists quickly fact-checked Mr. Giuliani’s false claim in a similar batch of tweets that there had been 63 murders.

The political discourse around gun violence — much like that around climate change — relies on data and would greatly benefit from more careful use of it.
Opening a file of Python code

```python
def parseDir(self, stanOutputDir):
    files = []
    for root, _, filenames in os.walk(stanOutputDir):
        for filename in fnmatch.filter(filenames, '*.x'):  # an extension
            files.append(os.path.join(root, filename))
    for f in files:
        doc_id = str(f.rfind('/') + 1):
        if doc_id in self.corpus.doc_idToDocs:
            self.docToSentenceTokens[doc_id] = self.parseFile(f)

# (1) reads stanford's output, saves it
# (2) aligns it w/ our sentence tokens

def parseFile(self, inputFile):
    sentenceTokens = defaultdict(lambda: defaultdict(int))
    tree = ET.ElementTree(file=inputFile)
    root = tree.getroot()
    document = root[0]
    sentences, _ = document
    for elem in corefs:
        print("el": elem)
        for section in elem:
            print("sec": section)
            for s2 in section:
                print("s2": s2)
    self.relationshipTypes = set()
    for elem in sentences:  # tree.iter(tag='sentence')
        sentenceNum = int(elem.attrib["id"])
        for section in elem:
            # process each token for the given sentence
            if section.tag == "tokens":  # the DependencyParse
                rootToken = StanToken(True, sentenceNum, 0, "ROOT", "ROOT" -1, ",", ",")
                sentenceTokens[sentenceNum][0] = rootToken
                for token in section:
                    tokenNum = int(token.attrib["id"])
                    word = ","
                    lemma = ","
                    startIndex = -1
                    endIndex = -1
                    s2 = ","
```

Atom

Text Edit
A REFRESHER — Python

• We choose to teach programming via the Python language
• Python is just the language of the words we choose to type
• We chose Python because:
  • it’s incredibly powerful (arguably the most robust language)
  • easy to read and write code
  • extensive set of libraries to help w/ doing technical stuff
• The skills you learn in this course (including writing Python code) are completely transferrable to other programming languages; after this course, it would be easy to write code in Java or R (just as Caroline, the TA. She took CS3). The core principles are the same!
A REFRESHER — Anaconda

- In order to run Python code (Python programs) that we write, we need to install software on our computer which knows how to understand and run Python code.

- Anaconda does this for us. Anaconda installs all necessary things so that we can write and run Python code.

- We didn’t have to use it; there are other ways to install Python, but it’s generally a very easy way to install Python.
A REFRESHER — The Terminal (aka Console)

- The terminal/console isn’t Python-specific! Inherently, has nothing to do with Python.

- It merely provides an alternative way to function with our computer, instead of the normal, graphical way with our mouse and clicking on folders and double-clicking programs to open them.

- Instead of using your computer via a mouse and clicking on pretty things, one could do most things while just using the Terminal/Console.

- Our Python programs we’ll create in this course don’t have graphical components that display stuff on the screen (e.g., Spotify), so it makes most sense to execute them from the Terminal.
Data Types

- Data Types
- Variables
- Operators
- Casting
Data Types

• all computer programs operate on data

• just like calculators (limited computers) do, but calculators operate only on numerical data

• our computer programs can operate on numerical data, text, and more.
Data Types

Primitive Data Types

• **Boolean Values:** only True or False

• **Numeric Values:** 0  -4  783910  33.3333333  -2.59

• **Strings (text):** “Hello”
  “Today, we heard from the Senate”
  “”
  “Welcome to CS3”
Data Types

- Boolean Values: only True or False
- Numeric Values: 0, -4, 783910, 33.3333333, -2.59
- Strings (text): “Hello”, “Today, we heard from the Senate”, “”, “Welcome to CS3”

**Boolean**

- Python’s type for booleans is `bool`
- Under the hood, your computer uses only 1 bit to store a boolean value
Data Types

• Boolean Values: only True or False
• Numeric Values: 0 -4 783910 33.3333333 -2.59
• Strings (text): "Hello"
  "Today, we heard from the Senate"
  ""
  "Welcome to CS3"

Integers
• Python’s type for integers is `int`
• Whole numbers only
• Calculations with integers are exact, except division
• Calculations with integers are crazy fast
• If your integer value is between ±9,223,372,036,854,775,807
• Under the hood, your computer uses 64 bits (8 bytes) to store an `int`
Data Types

- **Boolean Values:** only True or False
- **Numeric Values:** 0  -4  783910  33.3333333  -2.59
- **Strings (text):** "Hello"
  "Today, we heard from the Senate"
  ""
  "Welcome to CS3"

---

### Floating Point (numbers with decimals)

- Python’s type for these is called `float`
- Used for representing numbers with decimals
- Values are between a huge range: $-10^{307}$ to $10^{308}$
- Under the hood, your computer uses 64 bits (8 bytes) to store a `float`
Data Types

- **Boolean Values:** only True or False
- **Numeric Values:** 0 -4 783910 33.3333333 -2.59
- **Strings (text):** "Hello" "Today, we heard from the Senate" "" "Welcome to CS3"

---

**Floating Point (numbers with decimals)**

- Every calculation with floats is always approximate. Very close to correct, but unreliable after about 15 digits after the decimal. So, never check if two floats are equal.

  e.g.,

  ```python
  >>> 10/3
  3.3333333333333335
  ```

  Probably off
Data Types

■ Boolean Values: only True or False
■ Numeric Values: 0  -4  783910  33.3333333  -2.59
■ Strings (text): “Hello”

“Today, we heard from the Senate”

“”

“Welcome to CS3”

Strings (text)

■ Strings are used to represent words
■ A string is just a bunch of characters (a-Z, 0-9, etc)
■ Under the hood, the number of bytes your computer uses for a string depends on how long it is. (Uses 1 byte per character.)
Data Types

- Data Types
- Variables
- Operators
- Casting
Data Types

- Data Types
- Variables
- Operators
- Casting
Variables

• for computers programs to access and use any data, they must store the data somewhere, even if it’s a single, tiny piece of data (e.g., 1 number) and for a very brief time. Hence, why we have variables!

• **Variable** — something that represents a specific, stored piece of data. Each variable has a name (defined by the programmer), and that name is used to later reference/access/use the data.
Variables

- for computers programs to access and use any data, they must store the data somewhere, even if it’s a single, tiny piece of data (e.g., 1 number) and for a very brief time. Hence, why we have variables!

- **Variable** — something that represents a specific, stored piece of data. Each variable has a name (defined by the programmer), and that name is used to later reference/access/use the data.

  ```
  variable  value
  my_age   = 19
  ```
Variables

- for computers programs to access and use any data, they must store the data somewhere, even if it’s a single, tiny piece of data (e.g., 1 number) and for a very brief time. Hence, why we have variables!

- **Variable** — something that represents a specific, stored piece of data. Each variable has a name (defined by the programmer), and that name is used to later reference/access/use the data.

```plaintext
variable  value
my_age    = 19
my_school = “Brown University”
```
**Variable Assignment**

variable  value

\[ \text{result} = 2 + 2 \]
Variables

Variable Assignment

\[ \text{variable} \quad \text{value} \quad \text{expression} \]

\[ \text{result} = 2 + 2 \]

a series of computations, and it always results in a value (the output)
Variables

Variable Assignment

variable  value
result = 2 + 2

expression
a series of computations, and it always results in a value (the output)

assignment
assigns the value on the right to a memory address in the computer, and you can access it via the variable name result
Variable Assignment

result = 18 * 3 + 2
Variable Assignment

\[ \text{result} = 18 \times 3 + 2 \]
Variables

Variable Assignment

\[ \text{result} = 18 \times 3 + 2 \]

54
Variables

**Variable Assignment**

\[
\text{result} = 18 \times 3 + 2
\]

\[
\text{result} = 56
\]
Variables

Variable Assignment

\[
\text{result} = 18 \times 3 + 2
\]

stores 56 (in binary) in the computer’s memory somewhere, and you can always access this value via the variable named \text{result}. you could have named this variable whatever you want.

\text{result} \quad 0000000000000000000000000000000000000000000000000000000000111000
Variables

Variable Assignment

\[
\text{result} = 18 \times 3 + 2 \\
\text{another\_result} = 1 + \text{result}
\]
Variables

Variable Assignment

\[
\text{result} = 18 \times 3 + 2 \\
\text{another\_result} = 1 + \text{result}
\]
Variables

Variable Assignment

\[
\text{result} = 18 \times 3 + 2
\]

\[
\text{another\_result} = 1 + \text{result}
\]

1 + 56
Variable Assignment

\[ \text{result} = 18 \times 3 + 2 \]

\[ \text{another\_result} = 1 + \text{result} \]

57
Variables

Variable Assignment

\[
\text{result} = 18 \times 3 + 2
\]

\[
\text{another\_result} = 1 + \text{result}
\]

stores 57 (in binary) in the computer’s memory somewhere, and you can always access this value via the variable named \text{another\_result}. you could have named this variable whatever you want.
Variable Assignment

\[
\text{result} = 18 \times 3 + 2 \\
\text{another_result} = 1 + \text{result} \\
\text{name} = "\text{jim}\"
\]
Variable Assignment

\[
\text{result} = 18 \times 3 + 2
\]

\[
\text{another_result} = 1 + \text{result}
\]

\[
\text{name} = "\text{jim}"
\]

stores \text{jim} (in binary) in the computer’s memory somewhere, and you can always access this value via the variable named \text{name}. you could have named this variable whatever you want.
Variables

Execution

• a computer executes one line of a code at a time
• optionally assigns the computation’s output to a variable (if there’s an = sign).

1 18 * 3 + 2  \[\text{calculates 56 but doesn’t do anything with it}\]

• each line of code should do 1 thing, e.g.

1 \text{result} = 18 * 3 + 2
2 \text{another_result} = 1 + \text{result}

not

1 \text{result} = 18 * 3 + 2 \quad \text{another_result} = 1 + \text{result}
### Execution

- **Initializing a variable** — the first time you assign something to a variable; this is the *creation* of the variable.

- If you try to use a variable that you haven’t yet initialize (aka created/defined), your program will crash with an error.

1. `result = 18 * 3 + 2`
2. `another_result = 1 + results`
Variables

Execution

• **Initializing a variable** — the first time you assign something to a variable; this is the *creation* of the variable

• If you try to use a variable that you haven’t yet initialize (aka created/defined), your program will crash with an error.

```python
1   result = 18 * 3 + 2
2   another_result = 1 + results
```

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(results)
NameError: name 'results' is not defined
```
Data Types

- Data Types
- Variables
- Operators
- Casting
Data Types

- Data Types
- Variables
- Operators
- Casting
Operators

Mathematical Operators

result = a + b

operators can be:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Exponentiation: **
- Whole number result of division: //
- Remainder of division (modulo): %

a and b can be: numerical values (ints or floats) or text (strings)

but if you mix-and-match them, then you gotta be careful.
Mathematical Operators

```python
result = 5 + 3  # calculates 8
result = 5 / 3  # calculates 1.6666666666666667
result = 5 ** 3  # calculates $5^3$ which is 125
result = 5 // 3  # calculates 1
result = 5 % 3  # calculates 2
```
Mathematical Operators

- If an expression contains a `float` anywhere, the result will be a `float`

  \[ \text{result} = 5.2 + 3 \]  \[\text{calculates 8.2}\]

- If the result of division of two integers is not an integer, the result will be a float

  \[ \text{result} = 5 / 3 \]  \[\text{calculates 1.6666666666666667}\]
You can operate on any number or variable, including updating an existing variable’s value:

\[ \text{result} = \text{result} + 1 \]

adds 1 to the current value of \text{result}. standard way to represent a counter of something.
Mathematical Operators

You can operate on any number or variable, including updating an existing variable’s value:

\[
\text{result} = \text{result} + 1
\]

adds 1 to the current value of `result`. standard way to represent a counter of something.

For succinctness, you could alternatively type:

\[
\text{result} += 1
\]

same as above
Mathematical Operators

This nifty alternate version of writing can be used for all operations, e.g.,:

\[ \text{result} = \text{result} \times 2 \]

is equivalent to

\[ \text{result} *= 2 \]
String Operators

- String concatenation (aka combining words together):

\[
\text{result} = \text{“Red”} + \text{“Sox”} \quad \text{calculates “RedSox”}
\]

Mainly useful to concatenate when you’re combining text with numerical data (e.g., answers you care about) and you want to display it to the user.
String Operators

For example:

```
1 result = 6 / 3  # calculates 2.0. Remember
2 message = "The answer is " + result  # division always yields a float
```
Operators

String Operators

For example:

```python
1  result = 6 / 3  # calculates 2.0. Remember division always yields a float
2  message = "The answer is " + result
```

Although, UH-OH. We get an error.

```
>>> result = 6/3
>>> message = "The answer is " + result
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'float' object to str implicitly
```
String Operators

**TypeError:** The computation expects a value of a specific type, but received a different one instead.
What value does “result” contain?

1. \( \text{result} = 6 \div 3 \)
2. \( \text{age} = 20 \)
3. \( \text{result} = \text{result} + \text{age} \)
4. \( \text{result} \div= 2 \)
What value does “result” contain?

1. `result = 6 / 3`
2. `age = 20`
3. `result = result + age`
4. `result /= 2`

11.0
Data Types

- Data Types
- Variables
- Operators
- Casting
Data Types

- Data Types
- Variables
- Operators
- Casting
Casting

- Remember, mixing-and-matching numeric data (ints or floats) with text (strings) requires us to be careful.

- The computer doesn’t know what to do with these different data types, so it tells us that we must fix it.

- When trying to use different types of data together, which by default are incompatible, we must cast (convert) them (when possible).

```python
>>> result = 6/3
>>> message = "The answer is " + result
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'float' object to str implicitly
```
Casting

**Casting** — to change a particular data’s **type** of value(s)

```python
1       result  =  6 / 3
2       message  =  "The answer is " + str(result)
```

converts the float to a string!

```python
[>>>] result = 6/3
[>>>] message = "The answer is " + str(result)
[>>>] print(message)
The answer is 2.0
```
Casting — to change a particular data’s type of value(s)

```
1 result = 6 / 3
2 message = "The answer is " + str(result)
```

converts the float to a string!

```
>>> result = 6/3
>>> message = "The answer is " + str(result)
>>> print(message)
The answer is 2.0
```

btw, **print()** allows us to display to the screen the value of whatever is in the ()
Casting Examples

- We need to use `int()`, `float()`, `str()`, and `bool()`

```python
result = int("5")  # 5
result = float("5")  # 5.0
result = str(5)  # "5"
result = str(5.0)  # "5.0"
result = int("5.2")  # ValueError
result = int(float("5.2"))  # 5
result = float("h")  # ValueError
```
Casting Examples

• We need to use `int()`, `float()`, `str()`, and `bool()`

```python
result = int("5")
5
result = float("5")
5.0
result = str(5)
"5"
result = int("5.2")
ValueError
result = str(5.0)
"5.0"
result = int(float("5.2"))
5
ValueError: The computation expected a value with specific properties, but the value it received as input differed
```

```
result = float("h")
ValueError
```
Data Types

- Data Types
- Variables
- Operators
- Casting
Data Types

- Data Types
- Variables
- Operators
- Casting
Lab Time