Programming Principles in Python (CSCI 503/490)

Syntax & Types

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Administrivia

- Course Web Site
- TA: B V S Eswar Gottuparthi (Blackboard Collaborate)
- Syllabus
  - Plagiarism
  - Accommodations
- Assignments
- Tests: 2 (Sept. 27, Nov. 3) and Final (Dec. 6)
- Course is offered to both undergraduates (CS 490) and graduates (CS 503)
  - Grad students have extra topics, exam questions, assignment tasks
Using Python & JupyterLab on Course Server

- https://tiger.cs.niu.edu/jupyter/
- Login with your Z-ID (lowercase z)
- You should have received an email with your password
- Advanced:
  - Can add your own conda environments in your user directory
Using Python & JupyterLab Locally

- www.anaconda.com/download/
- Consider mamba (faster) and conda-forge
- Anaconda includes JupyterLab
- Use Python 3.10 (may have to install)
- Anaconda Navigator
  - GUI application for managing Python environment
  - Can install packages & start JupyterLab
- Can also use the shell to do this:
  - $ jupyter lab
  - $ conda install <pkg_name>
Zen of Python

- Written in 1999 by T. Peters in a message to Python mailing list
- Attempt to channel Guido van Rossum's design principles
- 20 aphorisms, 19 written, 1 left for Guido to complete (never done)
- Archived as PEP 20
- Added as an easter egg to python (import this)
- Much to be deciphered, in no way a legal document
- Jokes embedded
- Commentary by A.-R. Janhangeer
Explicit Code

• Goes along with complexity

• Bad:
  ```python
def make_complex(*args):
    x, y = args
    return dict(**locals())
  ```

• Good
  ```python
def make_complex(x, y):
    return {'x': x, 'y': y}
  ```
Don't Repeat Yourself

- "Two or more, use a for" [Dijkstra]
- Rule of Three: [Roberts]
  - Don't copy-and-paste more than once
  - Refactor into methods
- Repeated code is harder to maintain

Bad

```python
def load_file(file):
    pass
def get_cost(file):
    pass
f1 = load_file('f1.dat')
r1 = get_cost(f1)
f2 = load_file('f2.dat')
r2 = get_cost(f2)
f3 = load_file('f3.dat')
r3 = get_cost(f3)
```

Good

```python
def load_file(file):
    pass
def get_cost(file):
    pass
for i in range(1,4):
    f = load_file(f'f{i}.dat')
r = get_cost(f)
```
Multiple Types of Output

```
[2]: a = 12
    for i in range(3):
        print("Some output")
    plt.bar([1,2,3,4],[20,30,15,40])
    plt.show()
    a + 3
```

```
stdout
Some output
Some output
Some output
```

```
stderr
```

```
[2]: 15
```

```
output
```

```
[3]: 1 / 0

ZeroDivisionError Traceback (most recent call last)
<ipython-input-3-bc757c3fda29> in <module>
    ----> 1 1 / 0

ZeroDivisionError: division by zero
```
Assignment 1

- Due Friday
- Get acquainted with Python using notebooks
- Make sure to follow instructions
  - Name the submitted file a1.ipynb
  - Put your name and z-id in the first cell
  - Label each part of the assignment using markdown
  - Make sure to produce output according to specifications
- ipynb files are in a JSON format. Please maintain the .ipynb extension!
- Questions?
Quiz Thursday
Print function

- `print("Welcome, Jane")`
- Can also print variables:
  ```python
  first_name = "Jane"
  last_name = "Smith"
  print(last_name, first_name)
  ```
Python Math and String "Math"

- Standard Operators: +, -, *, /, %

- Division "does what you want" (new in v3)
  - 5 / 2 = 2.5
  - 5 // 2 = 2 # use // for integer division

- Shortcuts: +=, -=, *=

- No ++, --

- Exponentiation (Power): **

- Order of operations and parentheses: (4 - 3 - 1 vs. 4 - (3 - 1))
  - "abc" + "def"
  - "abc" * 3
Python Strings

• Strings can be delimited by single or double quotes
  - "abc" and 'abc' are exactly the same thing
  - Easier use of quotes in strings: "Joe's" or 'He said "Stop!"'
• Triple quotes allow content to go across lines and preserves linebreaks
  - """This is another
    string""
• String concatenation: "abc" + "def"
• Repetition: "abc" * 3
• Special characters: \n \t like Java/C++
Comments in Python

• # for single-line comments
  - everything after # is ignored
  - a = 3 # this is ignored
  - # this is all ignored

• Triple-quoted strings also used for comments (technically, any string can be)
  - A literal string without assignment, etc. is basically a no-op
  - """This is a string, often used as a comment"
  - """This string
    has multiple
    lines"

Python Variables and Types

- No declaration apart from assignment, no need for types
- Variables are names, not memory locations
  
  ```python
  a = 0
  a = "abc"
  a = 3.14159
  ```

- Strings are a type along with integer and floats
  - + containers (lists, dictionary)
  - + classes
Identifiers

• A sequence of letters, digits, or underscores, but…
• Also includes unicode "letters", spacing marks, and decimals (e.g. Σ)
• Must begin with a letter or underscore (_)
• Why not a number?
Identifiers

- A sequence of letters, digits, or underscores, but…
- Also includes unicode "letters", spacing marks, and decimals (e.g. Σ)
- Must begin with a letter or underscore (_)
- Why not a number? Ambiguity, 8j is a complex number, 8e27 is a float
- Case sensitive (a is different from A)

Conventions:
- Identifiers beginning with an underscore (_) are reserved for system use
- Use underscores (a_long_variable), not camel-case (aLongVariable)
- Keep identifier names less than 80 characters
- Cannot be reserved words
Reserved Words and Reassigning builtins

- Some words cannot serve as identifiers (called keywords in Python)
  - `import keyword`
    ```python
    keyword.kwlist
    ```
  - `['False', 'None', 'True', 'and', 'as', 'assert', 'async',
    'await', 'break', 'class', 'continue', 'def', 'del',
    'elif', 'else', 'except', 'finally', 'for', 'from',
    'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
    'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
    'with', 'yield']`
  - `False = True` # SyntaxError

- Some other words (python's builtins) can, but this can cause problems
  - `int = 34`
    ```python
    int("12")` # TypeError
    ```
Programming Principle: Use Meaningful Identifiers

- Show intention:
  - Bad: var34
  - Good: time_difference

- Easy pronunciation: Not egészségedre (perhaps ok if you're Hungarian)

- Simple but technical:
  - Bad: in_order_list_of_jobs
  - Good: job_queue

- Be consistent:
  - Bad: user_list and groups
  - Good: user_list and group_list
Types

• Don't worry about types, but think about types
• Variables can "change types"
  - \( a = 0 \)
  - \( a = "abc" \)
  - \( a = 3.14159 \)
• Actually, the **name** is being moved to a different value
• You can find out the type of the value stored at a variable \( v \) using `type(v)`
• Some literal types are determined by subtle differences
  - 1 vs 1. (integer vs. float)
  - 1.43 vs 1.43j (float vs. imaginary)
  - '234' vs b'234' (string vs. byte string)
Type Conversion

- Python converts integers to floats when types are mixed
  - 1 + 3.4 # evaluates to 4.4 (float)

- Functions can return different types than inputs
  - round(3.9) # evaluates to 4 (int)

- Can do explicit type conversion
  - int(3.9) # evaluates to 3 (int)
  - float(123) # evaluates to 123. (float)
  - int("123") # evaluates to 123 (int)
  - str(123) # evaluates to "123" (string)
Numeric Precision

- Integers have infinite precision and are as big as you want them
  - 93326215443944152681699238856266700490715968264381621468592
  96389521759999322991560894146397615651828625369792082722375
  8251185210916864000000000000000000000000

- Floats do not have infinite precision but still hold large numbers (double-precision)
  - 9.33262154439441e+157
  - Python keeps 17 significant digits
  - Python by default only prints up to 12 (many times less)

- Python has support for infinite precision (Decimal)

- How might this work; how could you store a floating point number with infinite precision using python?
Expression Rules

- Involve
  - Literals (1, "abc"),
  - Variables (a, my_height), and
  - Operators (+, -, *, /, //, **)

- Spaces are irrelevant within an expression
  - a + 34 # ok

- Standard precedence rules
  - Parentheses, exponentiation, mult/div, add/sub
  - Left to right at each level

- Also boolean expressions
Assignment

- The `=` operator
- Can assign a literal, another variable, or any expression
  - `a = 34`
  - `b = a`
  - `c = (a + b) ** 2`
- Cannot use this operator in the middle of an expression, like in C++
- However, Python 3.8 added a new operator (the "walrus") that allows this
Assignment

- Other languages: set aside memory space for value and give that space a name; space can be updated with a new value

```plaintext
int x = 42;
```

```plaintext
x = x + 1;
int y = x;
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>43</td>
</tr>
</tbody>
</table>
```
Assignment

- Python variables are actually **pointers** to objects (names for values)

\[
x = 42
\]

\[
x = x + 1
y = x
\]
Augmented Assignment

• Shorthand for mutation of a variable's value stored back in the same variable
  
i += 1 # same thing as i = i + 1

• +=, -=, *=, /=, //=, **=

• Python does not have ++ or --
Simultaneous Assignment

• Feature that doesn't appear in many other languages
• Allows multiple expressions to be assigned to different variables with one assignment
  - \( a, b = 34 \times 2, 400 / 24 \)
• Commas separate the variables and expressions
• Most useful for swapping variables
  - \( a, b = b, a \)
• How does this usually work?
Simultaneous Assignment

• In most languages, this requires another variable
  - `x_old = x`  
  `x = y`  
  `y = x_old`

• Simultaneous assignment leaves less room for error:
  - `x, y = y, x`

• Also useful for unpacking a collection of values:
  - `dateStr = "03/08/2014"`  
    `monthStr, dayStr, yearStr = dateStr.split("/")`
Assignment Expressions

• AKA the "walrus" operator :=
• Names a value that can be used but also referenced in the rest of the expression
• (my_pi := 3.14159) * r ** 2 + a ** 0.5/my_pi
• Use cases: if/while statement check than use, comprehensions
• Supported in Python 3.8+
Assignment Expressions

• Contentious discussion on adding to the language
  - "There should be one-- and preferably only one --obvious way to do it"
  - Leads to different coding styles
• Adopted, and community moving on to best practices