

# Objectives

- Review
- Lab 1
  - Linux practice
  - Programming practice
    - Print statements
    - Numeric operations, assignments
    - Input

# Lab 0 Feedback

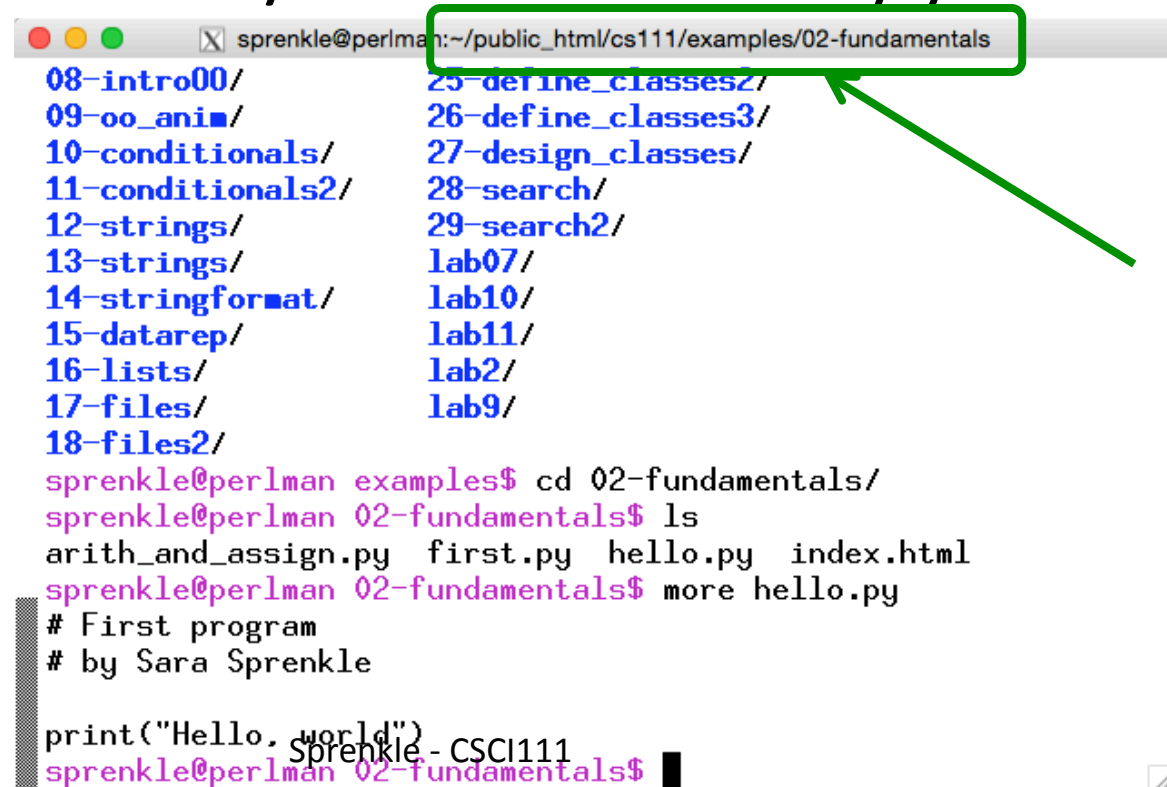
- Overall, did well
  - Lost points because didn't check work
    - E.g., broken Web page links, not including required text
  - Generally, lab grades should be high
- Interesting article links!
  - Consider reviewing for extra credit
- Sakai extra credit Easter egg
  - Great fun facts!

# Lab 0 Feedback

- If there were any issues with your web page, go back and fix them first.
  - We can help!
  - Goal: Make sure you're set up for the semester

# Lab 1: Linux Practice

- Review your notes, handouts from last lab
- Setting up directories
  - Make the directory, copy files
- Note: terminal tells you which directory you're in



A terminal window screenshot showing a user named 'sprenkle' at a machine named 'perlman'. The current directory is '~/.public\_html/cs111/examples/02-fundamentals', which is highlighted with a green box and an arrow. The terminal displays a list of subdirectories in two columns: 08-intro00/, 09-oo\_anim/, 10-conditionals/, 11-conditionals2/, 12-strings/, 13-strings/, 14-stringformat/, 15-datarep/, 16-lists/, 17-files/, 18-files2/, 25-define\_classes2/, 26-define\_classes3/, 27-design\_classes/, 28-search/, 29-search2/, lab07/, lab10/, lab11/, lab2/, and lab9/. The user then runs 'cd 02-fundamentals/' and 'ls', showing files: arith\_and\_assign.py, first.py, hello.py, and index.html. Finally, the user runs 'more hello.py', displaying the contents of the file: '# First program', '# by Sara Sprenkle', and 'print("Hello, world")'. The terminal prompt is 'sprenkle@perlman 02-fundamentals\$'.

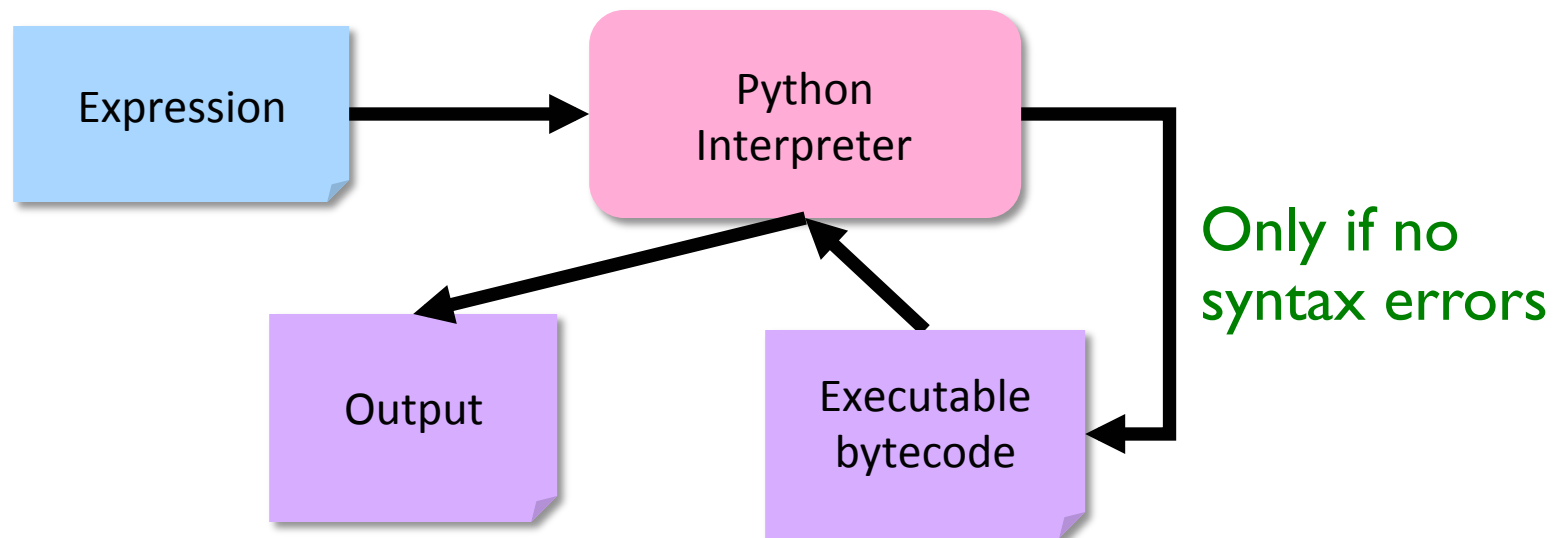
```
sprenkle@perlman:~/public_html/cs111/examples/02-fundamentals
08-intro00/      25-define_classes2/
09-oo_anim/      26-define_classes3/
10-conditionals/ 27-design_classes/
11-conditionals2/ 28-search/
12-strings/      29-search2/
13-strings/      lab07/
14-stringformat/ lab10/
15-datarep/      lab11/
16-lists/        lab2/
17-files/        lab9/
18-files2/

sprenkle@perlman examples$ cd 02-fundamentals/
sprenkle@perlman 02-fundamentals$ ls
arith_and_assign.py first.py hello.py index.html
sprenkle@perlman 02-fundamentals$ more hello.py
# First program
# by Sara Sprenkle

print("Hello, world")
sprenkle@perlman 02-fundamentals$
```

# Python Interpreter

1. Validates Python programming language expression(s)
  - Enforces Python syntax rules
  - Reports syntax errors ← Have a lot of these early on!
2. Executes expression(s)



# Two Modes to Execute Python Code

- **Interactive:** using the interpreter
  - Try out Python expressions
- **Batch:** execute *scripts* (i.e., files containing Python code)
  - What we'll write usually

# Python Interpreter: Interactive Mode

Run by typing `python3` in terminal

```
sprengle@perlman private$ python3
Python 3.4.1 (default, Sep 24 2015, 20:41:10)
[GCC 4.9.2 20150212 (Red Hat 4.9.2-6)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 3
3
>>> 4+5
9
>>> 1-7
-6
>>> "word"
'word'
>>> word
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'word' is not defined
>>> print 4+5
File "<stdin>", line 1
  print 4+5
    ^
SyntaxError: invalid syntax
>>> print(4+5)
9
^_^
```

Python displays the result

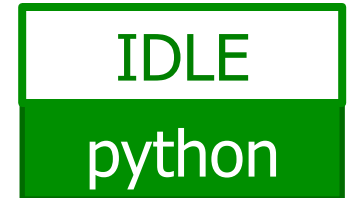
Type in the expression

**Error Message:**  
We'll talk more later about why this is an error

`print`: Special *function* to display output

# IDLE Development Environment

- IDLE development environment
  - Runs on top of Python interpreter
  - Command: `idle3 &`
    - `&` Runs command in “background” so you can continue to use the terminal



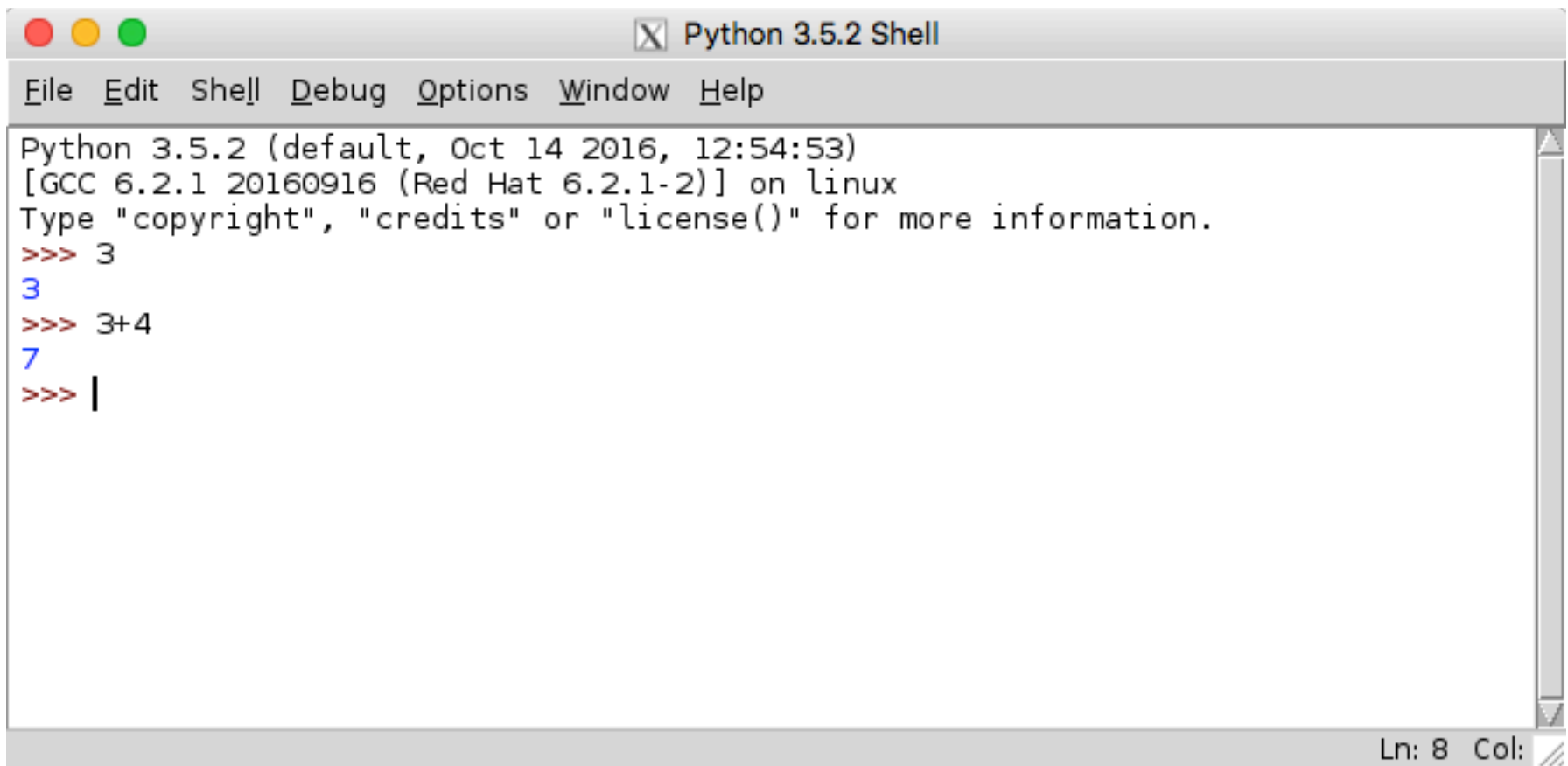
Since our programming language is named after Monty Python, what is the development environment named after?

- Can use IDLE to
  - Run Python in **interactive** mode
  - Write and execute scripts in **batch** mode



# IDLE

- IDLE first opens up a Python shell
  - i.e., the Python interpreter in interactive mode



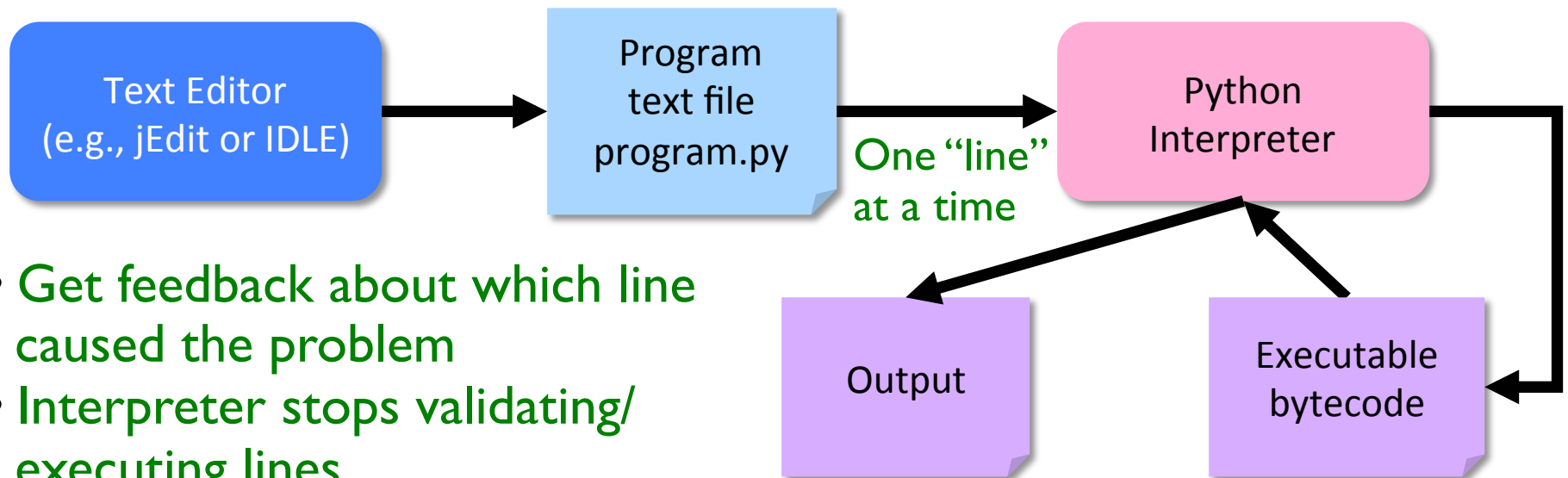
```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (default, Oct 14 2016, 12:54:53)
[GCC 6.2.1 20160916 (Red Hat 6.2.1-2)] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 3
3
>>> 3+4
7
>>> |
Ln: 8 Col: 9
```

# Your Turn in Interactive Mode...

- Run `idle3` or `python3`
- Enter the following expressions and see what Python displays:
  - `3`
  - `4 * -2`
  - `-1+5`
  - `2 +`
  - `print("Hello!")`
- If you used `python3`, to quit the interpreter, use Control-D

# Batch Mode

1. Programmer types a **program/script** into a **text editor** (jEdit or IDLE).
2. An **interpreter** turns each expression into **bytecode** and then executes each expression



- **Get feedback about which line caused the problem**
- **Interpreter stops validating/ executing lines**

# Example Python Script

Text file named: hello.py

```
# Program that prints out "Hello, world!"  
# by Sara Sprenkle, 09/17/2017  
  
print("Hello, world!")
```

Print statement



- What does this program do?
  - Validate your guess by executing the program
    - Go into /csdept/courses/cs111/lab1 directory
    - `python3 hello.py`

# Example Python Script

```
# Program that prints out "Hello, world!"  
# by Sara Sprenkle, 09/17/2017  
  
print("Hello, world!")
```

} Documentation  
-- good style

- Only **Hello, world!** is printed out
- Python ignores everything after the “#”
  - Known as “**comments**” or, collectively, as **documentation**

Your program should *always* start with a high-level description of what the program does, your name, and the date the program was written

# IDLE

- In IDLE, under the File menu
  - Use **New File or Open**, as appropriate, to open a window so that you can write your Python script.

# Recap: Executing Python

- Interactive Mode
  - Try out expressions
  - `python3`
- Batch Mode
  - Execute Python scripts
  - `python3 <pythonscript>`
- **IDLE** combines these two modes into one integrated development environment

# Review

- How do we display output?
- What are the data types available in Python?
- How should we name variables?
- How do we assign values to variables?



# Review

- How do we get input from the user?
  - Numerical vs textual (string)
- What is our process for solving problems?
- What is the two-part verification process we need to do after we implement a program?

# Improving average2.py

- Get user input!
- Keep in mind: Good development process
  1. “hardcode” your values
  2. Work on the program
  3. Add user input

# Another Example:

## Restricting User's Inputs

```
>>> x = 7
>>> yourVal = input("My val is: ")
My val is: x
>>> print(yourVal)
x
```

# Another Example:

## Restricting User's Inputs

```
>>> x = 7
>>> yourVal = input("My val is: ")
My val is: x
>>> print(yourVal)
x
>>> yourVal = eval(input("My val is: "))
My val is: x
>>> print(yourVal)
7
What happened here?
>>> yourVal = int(input("My val is: "))
My val is: x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'x'
```

# Recap: Programming Fundamentals

- Most important data types (for us, for now):  
**int, float, str, bool**
  - Use these types to represent various information
- Variables have identifiers, (implicit) types
  - Should have “good” names
  - Names: start with lowercase letter; can have numbers, underscores
- Assignments
  - $x = y$  means “x set to value y” or “x is assigned value of y”
  - Only variable on LHS of statement changes

# Review: Assignment statements

- Assignment statements are NOT math equations!

```
count = count + 1
```

- These are commands!

```
x = 2
```

```
y = x
```

```
x = x + 3
```

What is the value of y?

# Numeric Arithmetic Operations

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder ("mod")
**	Exponentiation (power)

**Remember PEMDAS**

# Review: Arithmetic Operations

Symbol	Meaning	Associativity
+	Addition	Left
-	Subtraction	Left
*	Multiplication	Left
/	Division	Left
%	Remainder ("mod")	Left
**	Exponentiation (power)	Right

Precedence rules: P E - DM% AS

↑  
negation

*Associativity* matters when you have the same operation multiple times



# Review: Two Division Operators

## / Float Division

- Result is a **float**
- Examples:
  - $6/3 \rightarrow 2.0$
  - $10/3 \rightarrow 3.33333333333333333335$
  - $3.0/6.0 \rightarrow 0.5$
  - $10/9 \rightarrow 1.9$

## // Integer Division

- Result is an **int**
- Examples:
  - $6//3 \rightarrow 2$
  - $10//3 \rightarrow 3$
  - $3.0//6.0 \rightarrow 0$
  - $10//9 \rightarrow 1$

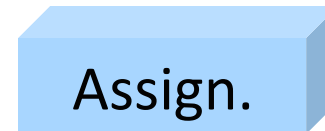
# Programming Building Blocks

- Each type of statement is a building block

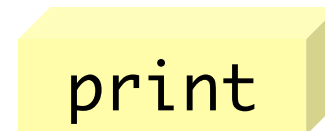
- Initialization/Assignment

- So far: Arithmetic

- Print



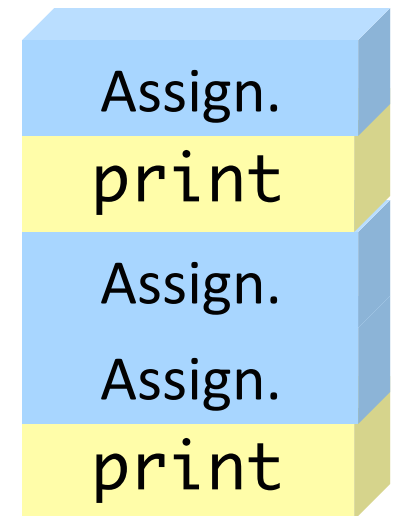
Assign.



print

- We can combine them to create more complex programs

- Solutions to problems

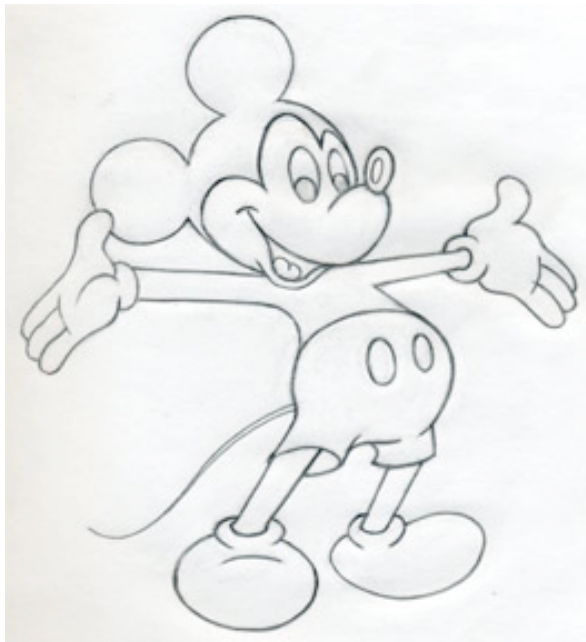


Assign.  
print  
Assign.  
Assign.  
print

# Formalizing Process of Developing Computational Solutions

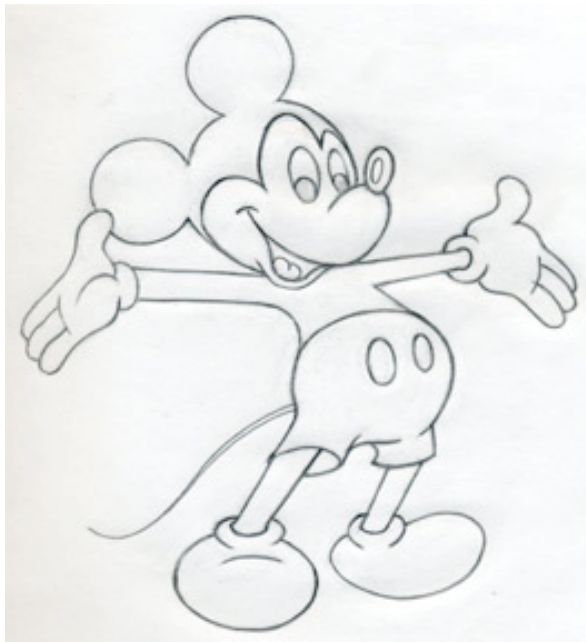
1. Create a sketch of how to solve the problem  
(the algorithm)

Use comments to describe the steps



# Formalizing Process of Developing Computational Solutions

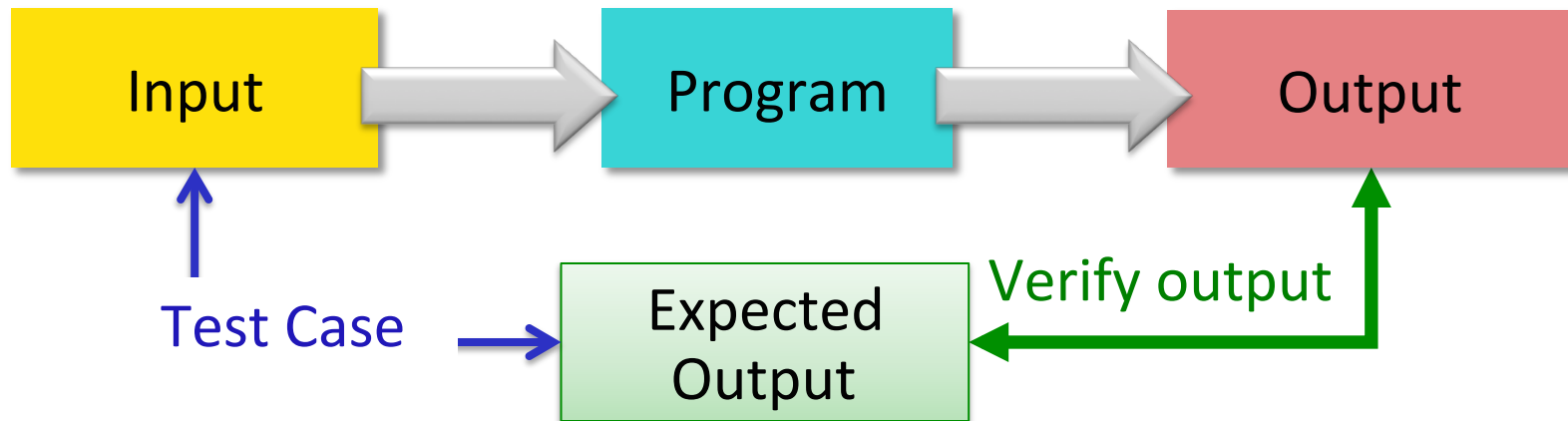
1. Create a sketch of how to solve the problem  
(the algorithm)
2. Fill in the details in Python



# Errors

- Sometimes the program doesn't work
- Types of programming errors:
  - Syntax error
    - Interpreter shows where the problem is
  - Logic/semantic error
    - `answer = 2+3`
    - No, answer should be `2*3`
  - Exceptions/Runtime errors
    - `answer = 2/0`
    - Undefined variable name

# Testing Process

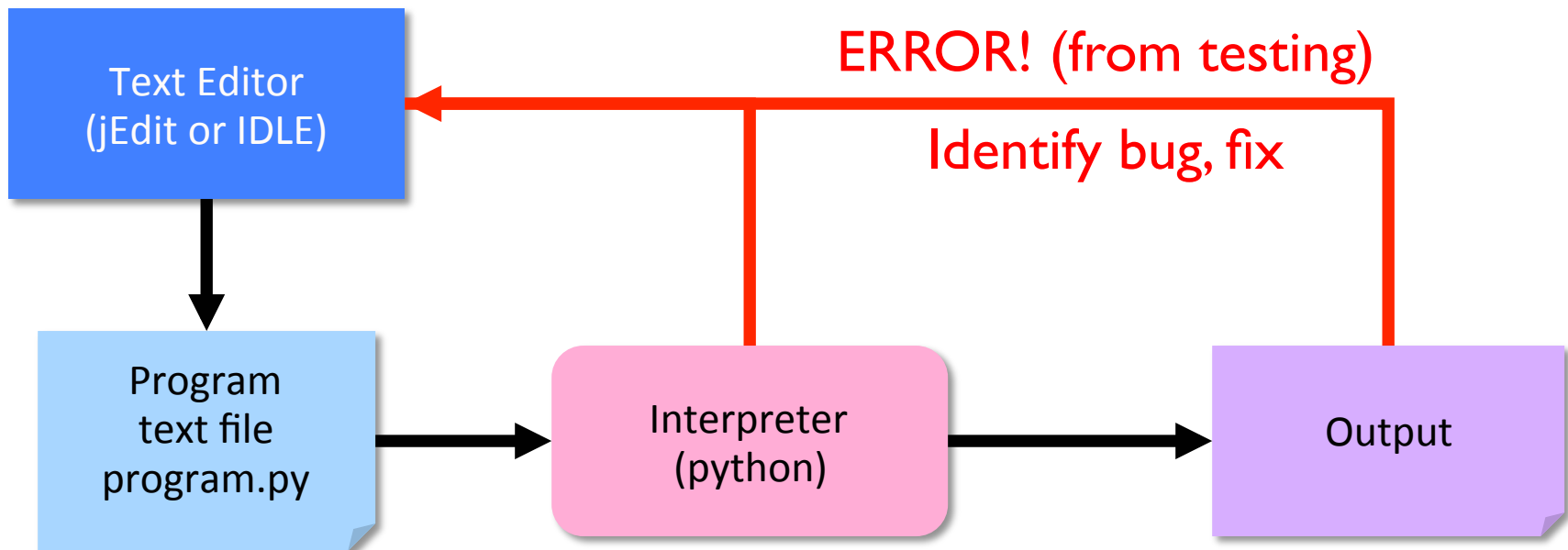


- **Test case:** **input** used to test the program, **expected output** given that input
- Verify if **output** is what you expected

If output is not what you expect...

# Debugging

- After identifying errors during *testing*
- Identify the problems in your code
  - Edit the program to fix the problem
  - Re-execute/test until all test cases pass
- The error is called a “bug” or a “fault”
- Diagnosing and fixing error is called ***debugging***



# Design Patterns

- General, repeatable solution to a commonly occurring problem in software design
  - Template for solution



# Review: Design Patterns

- General, repeatable solution to a commonly occurring problem in software design

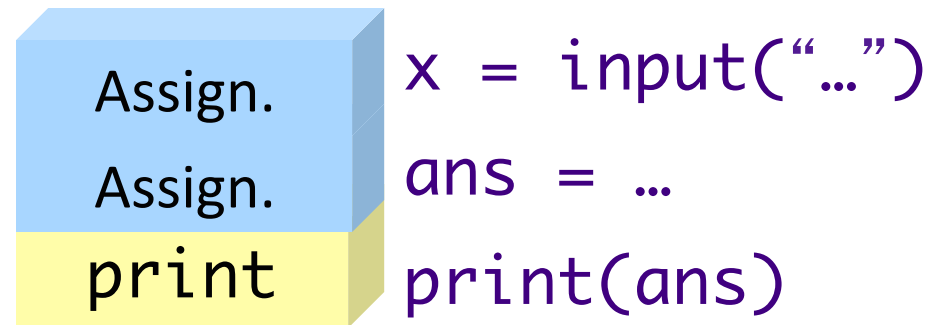
- Template for solution

- Example (Standard Algorithm)

- Get input from user

- Do some computation

- Display output



# Lab 1 Expectations

- Comments in programs
  - High-level comments, author
  - Notes for your algorithms, implementation
- Nice, readable, understandable output
  - User running your program needs to understand what the program is saying
- Honor System
  - Pledge the Honor Code on printed sheets

Reintroduce lab assistants

# Lab 1: Programming Practice

- After the warm up problems
- Name program files **lab1.n.py**, where  $n$  is the problem you're working on
- After completed, demonstrate that your program works
  1. Close IDLE/Python interpreter, rerun program
    - Get rid of the output from when you were developing/debugging (“scratch work”)
  2. Save output for each program in file named **lab1.n.out** where  $n$  is the problem you're working on

# Lab 1 Expectations: Example Output

- For programs that take user input, run **multiple times** to demonstrate that the program works.
- Resulting output should be saved in a `.out` file
- For example, what would the output for the `average2.py` program look like?

# Lab 1 Submission

- Electronic as well as printed
  - I can execute your program, help find mistakes
  - Copy your lab directory into your turnin directory
- Instructions are in the lab

# Honor

- You may discuss programming assignments *informally* with other students
  - Sharing the **code** is an honor violation
  - Do **not share** your password
- You should know where to draw the line between legitimate outside assistance with course material and outright cheating
  - Students who obtain too much assistance without learning the material ultimately cheat themselves
- If you have any uncertainty about what this means, consult with me before you collaborate.

# Honors System: Rules of Thumb

- Discussion of problems/programs - OK
  - Clarification questions
  - Algorithm discussion (on paper, board)
- Debugging help
  - Programmer always “owns” keyboard, mouse
  - Helper can read other’s program/debug/help, up to 5 minutes
    - Ask TA or me or email me for problems that require more time