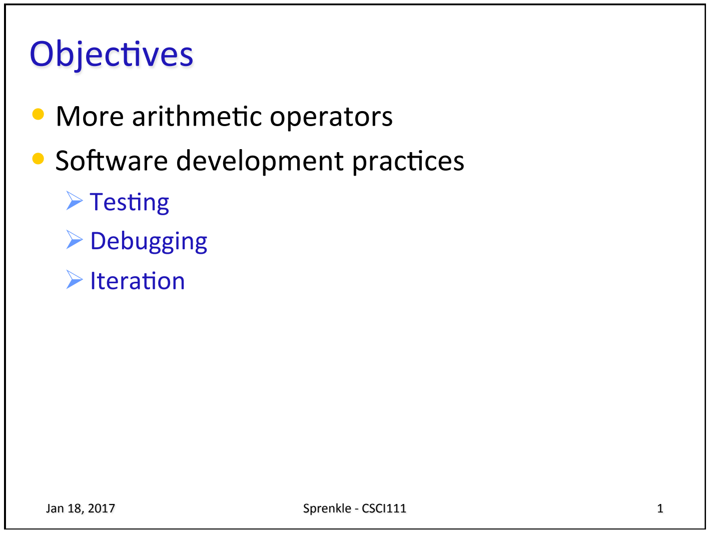
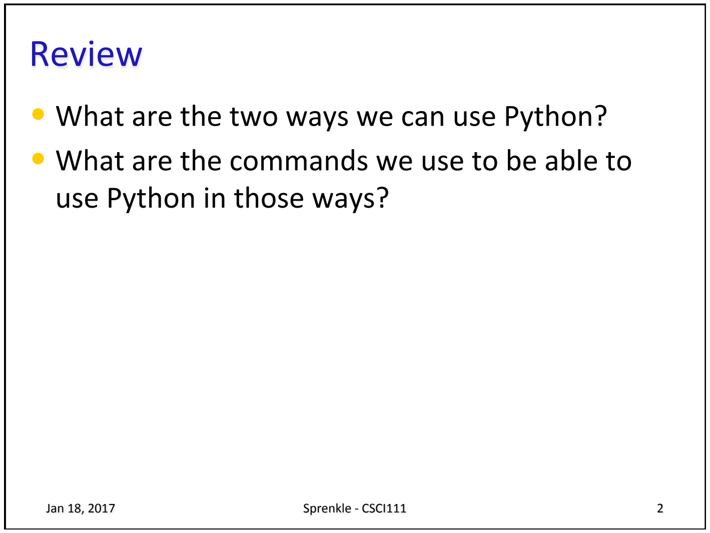


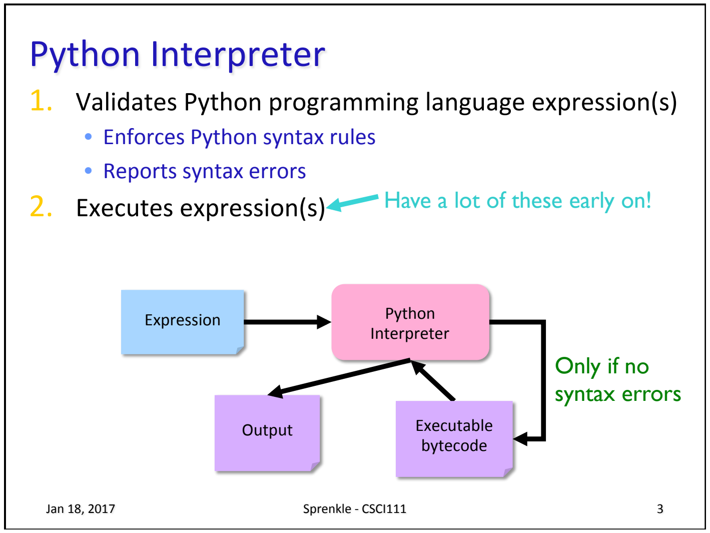
## Objectives

- More arithmetic operators
- Software development practices
  - Testing
  - Debugging
  - Iteration



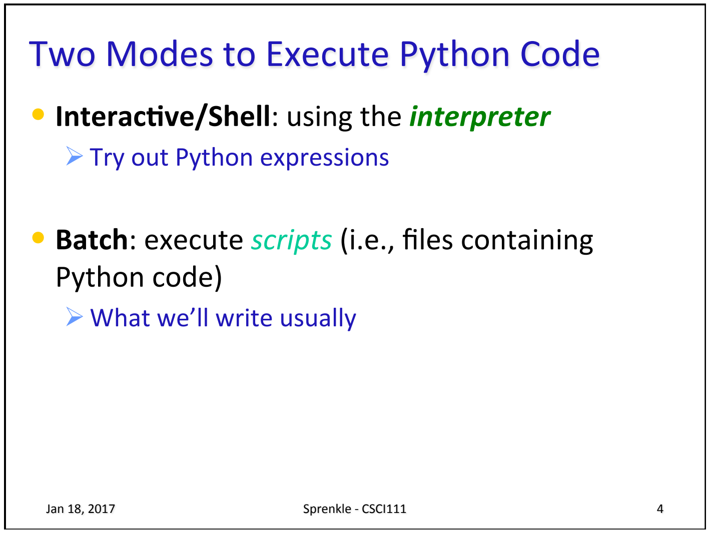
## Review

- What are the two ways we can use Python?
- What are the commands we use to be able to use Python in those ways?



## Python Interpreter

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



## Two Modes to Execute Python Code

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

## Lessons from Lab

- Look at examples!
  - “I was able to do this in that other program. How did I do that?”
- Explore!
  - Try things out in interactive mode
  - Then, put the ones that work into a script/program
- Follow all of the directions!

## Review: NOT Math Class

- Need to write out all operations explicitly
  - In math class, a (b+1) meant $\underbrace{a\ *\ (b+1)}_{\text{Write this way in Python}}$

## Review

- What are Python’s primitive data types and what do they represent?
- How do we name variables?
  - What is another word for “variable name” in programming?
- How do we give variables values?

## Parts of an Algorithm

- Input, Output
- Primitive operations
  - What data you have, what you can do to the data
- Naming
  - Identify things we’re using
- Sequence of operations
- Conditionals
  - Handle special cases
- Repetition/Loops
- Subroutines
  - Call, reuse similar techniques

## Two Division Operators

**/ Float Division**

- Result is a `float`
- Examples:
  - 6/3 → 2.0
  - 10/3 → 3.3333333333333335
  - 3.0/6.0 → 0.5
  - 19/10 → 1.9

**// Integer Division**

- Result is an `int`
- Examples:
  - 6//3 → 2
  - 10//3 → 3
  - 3.0//6.0 → 0.0
  - 19//10 → 1

Integer division is the default division used in most programming languages

## Integer Division Practice

- `a = 12//4`
- `4 // 6 * 5.0`
- `b = 6/12`
- `6.0//12`
- `z = a / b`

## More on 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. It tells you where you should start computing.

## Math Practice

```
5 + 3 * 2
2 * 3 ** 2
-3 ** 2
2 ** 3 ** 3
```

How should we verify our answers?

## Modulo Operator: %

- Modular Arithmetic: Remainder from division
  - `x % y` means the remainder of x//y
  - Read as "x mod y"
- Example: `6 % 4`
  - Read as "six mod four"
  - 6//4 is 1 with a remainder of 2, so 6%4 evaluates to 2
- Works only with integers
  - Typically just positive numbers
- Precedence rules: P E - DM% AS

## Modulo Practice

- `7 % 2`
- `3 % 6`
- `6 % 2`
- `7 % 14`
- `14 % 7`
- `6 % 0`

## Brainstorm

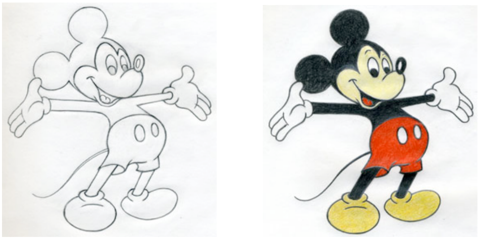
- What useful thing does % 10 do?
  - 3 % 10 =
  - 51 % 10 =
  - 40 % 10 =
  - 678 % 10 =
  - 12543 % 10 =
- What useful thing does // 10 do (integer division)?
  - 3 // 10 =
  - 51 // 10 =
  - 40 // 10=
  - 678 // 10 =
  - 12543 // 10 =
- What useful thing does % 2 do?

## Formalizing Process of Developing Computational Solutions

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

## 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

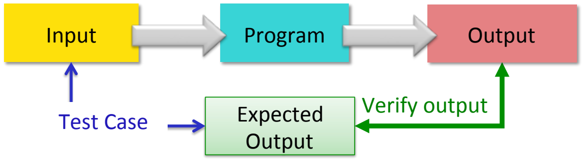


## Errors

- 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

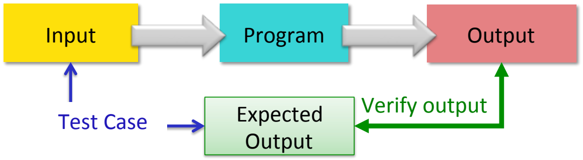
Expose errors when **Testing**

## Testing Process



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

## Testing Process

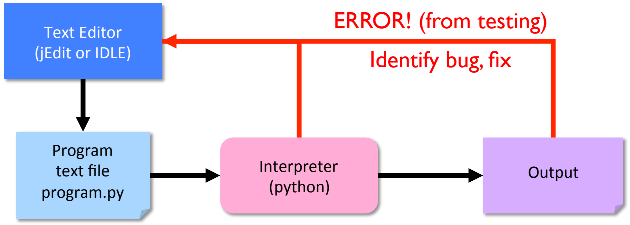


- Need ***good*** **test cases** to help determine if program is correct
  - Tester plays devil's advocate
  - Want to expose **all** errors!
  - Find before customer/professor!

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***



## 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
3. Test the Python program with *good* test cases
   a. If errors found, debug program
   b. Repeat step 3

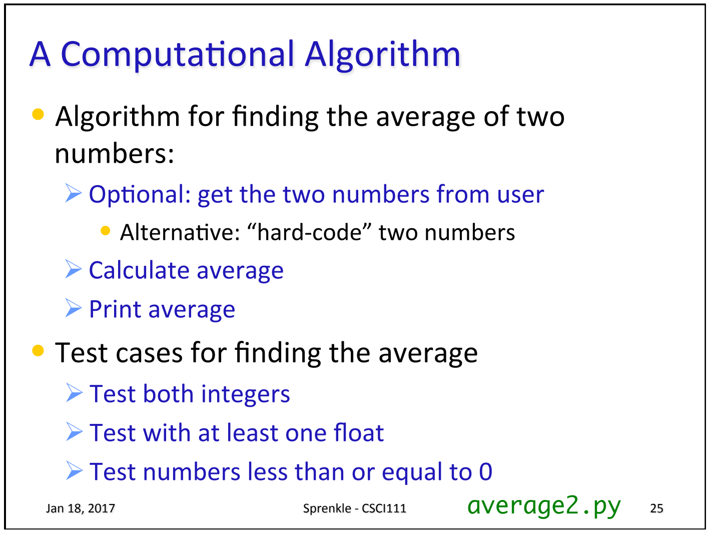
## Practice: A Computational Algorithm

- Find the average of two numbers

## Practice: A Computational Algorithm

- Find the average of two numbers
- Test cases:

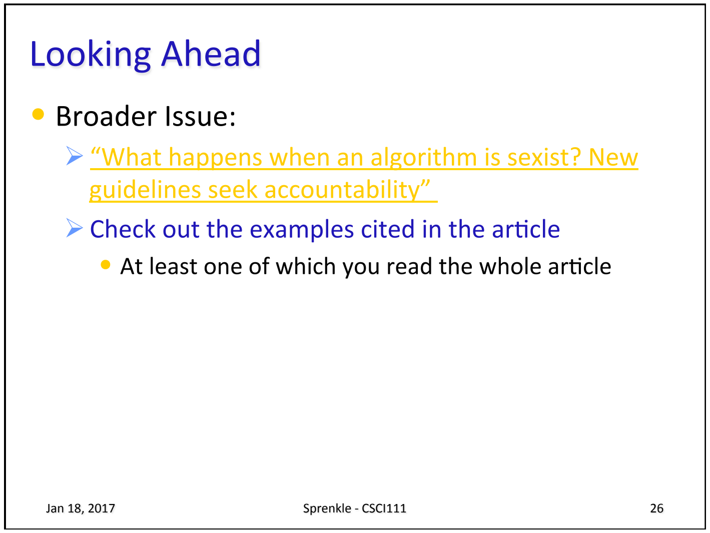
| **Input** | | |
|---|---|---|
| num1 | num2 | **Expected Output** |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |



## A Computational Algorithm

- Algorithm for finding the average of two numbers:
  - Optional: get the two numbers from user
    - Alternative: “hard-code” two numbers
  - Calculate average
  - Print average
- Test cases for finding the average
  - Test both integers
  - Test with at least one float
  - Test numbers less than or equal to 0

average2.py



## Looking Ahead

- Broader Issue:
  - “What happens when an algorithm is sexist? New guidelines seek accountability”
  - Check out the examples cited in the article
    - At least one of which you read the whole article