## Objectives

- Developing Programs
- Input
- Intro to design patterns

## Review

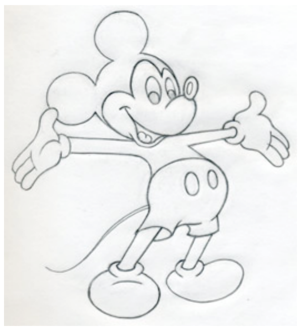
- What kinds of division are there?
    - How are they different?
- What does % mean?

## 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?

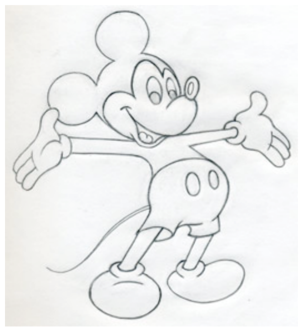
## Formalizing Process of Developing Computational Solutions

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

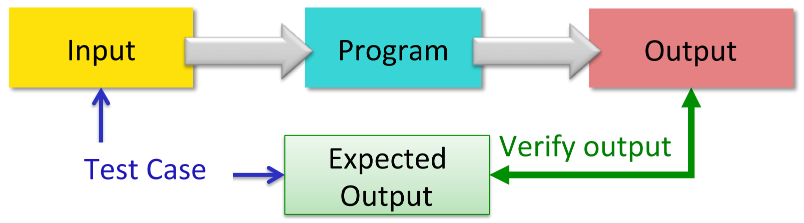


## Errors

- 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

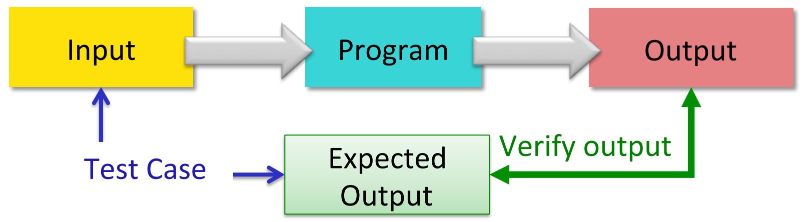
Expose errors when **Testing**

## Testing Process



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

## Testing Process

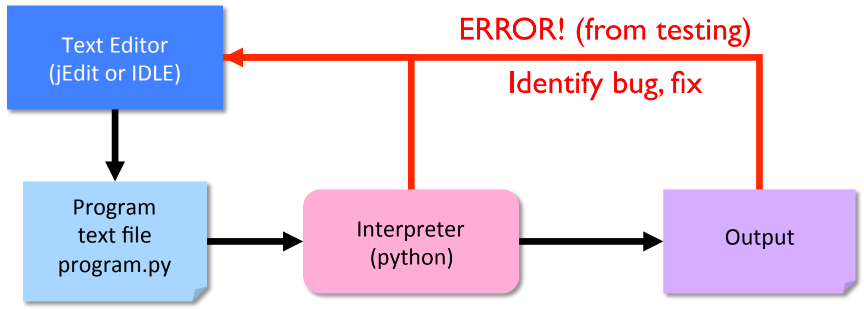


- Need ***good*** **test cases** to help determine if program is correct
  - Tester plays devil's advocate
  - Want to expose **all** errors!
  - Find before customer/professor!

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`

## Review: 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

Approximately Chapter 3 of text book

## Good Development Practices

- Design the algorithm
  - Break into pieces
- **Implement** *and* **Test** each piece *separately*
  - Identify the best pieces to make progress
  - Iterate over each step to improve it
- Write comments FIRST for each step
  - Elaborate on what you're doing in comments when necessary

`average2.py`

## When to Use Comments

- Document the author, high-level description of the program at the top of the program
- Provide an outline of an algorithm
  - Separates the steps of the algorithm
- Describe difficult-to-understand code

## Trick: Type Conversion

- You can convert a variable's type
  - Use the type's ***constructor***

| Conversion Function/Constructor | Example | Value Returned |
|---|---|---|
| `int(<number or string>)` | `int(3.77)`<br>`int("33")` | 3<br>33 |
| `float(<number or string>)` | `float(22)` | 22.0 |
| `str(<any value>)` | `str(99)` | "99" |

## 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

## Interactive Programs

2.8 in Text Book

- Meaningful programs often need input from users
- Demo: `input_demo.py`

## Getting Input From User

- `input` is a ***function***
  - **Function**: A command to do something
    - A “subroutine”
- Syntax:
  - `input(<string_prompt>)`
- Semantics:
  - Display the prompt `<string_prompt>` in the terminal
  - Read in the user’s input and *return* it as a string/text

## Getting Input From User

- Typically used in assignments
- Examples:

Prompt displayed to user

$$\texttt{name=input(}\underbrace{\texttt{"What is your name? "}}_{}\texttt{)}$$

  - name=input("What is your name? ")
    - name is assigned the string the user enters
  - width=eval(input("Enter the width:"))
    - **What the user enters is evaluated (as a number) and assigned to `width`**
    - Use `eval` function because expect a number from user

What do you think the code looks like for `input_demo.py`?

## Getting Input from User

```
color = input("What is your favorite color? ")
```

Semantics: Sets the variable `color` to the user's input

**Terminal:**

Grabs every character up to the user presses "enter"

```
> python3 input_demo.py
What is your favorite color? blue
Cool!  My favorite color is _light_ blue !
```

`input_demo.py`

## Example Using Type Conversion

- May want to restrict the type of values that a user enters
- For example, a user's age should be an integer

```
str_age = input("What is your age? ”)
int_age = int(str_age)

print("Your age is", int_age)
```

Converts age to an integer

Ideally, we'd tell the user that we made a change to their input, but we don't know how to do that yet.

## 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
>>> 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'
```

What happened here?

## Identify the Parts of a Program

```
# Demonstrate numeric and string input
# by Sara Sprenkle for CS111
#

color = input("What is your favorite color? " )
print("Cool!  My favorite color is _light_", color, "!")

rating = eval(input("On a scale of 1 to 10, how much do
you like Ryan Gosling? "))
print("Cool!  I like him", rating*1.8, "much!")
```

Identify the comments, variables, functions, expressions, assignments, literals

input_demo.py

## Identify the Parts of a Program

```
# Demonstrate numeric and string input
# by Sara Sprenkle for CS111
#

color = input( “What is your favorite color? " )
print("Cool!  My favorite color is _light_" , color, "!")

rating = eval(input( "On a scale of 1 to 10, how much do
you like Ryan Gosling? ” )
print("Cool!  I like him” , rating*1.8, "much!")
                               expression
```

Identify the comments, variables, functions, expressions, assignments, literals

## Improving `average2.py`

- With what we just learned, how could we improve `average2.py`?
- Example of suggested approach to development
  - Input is going to become fairly routine.
  - Wait on input until you have figured out the rest of the program/problem.

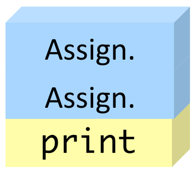
## Design Patterns

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

## 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



```
x = input("…")
ans = …
print(ans)
```

## Looking Ahead

- Lab 1 Tomorrow!
  - Lab 1 Prep due tomorrow before lab