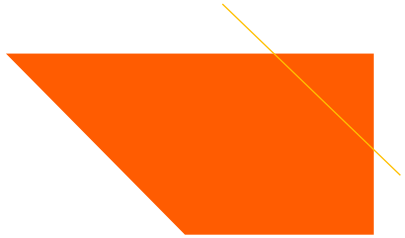
Introduction to Problem Solving in Python

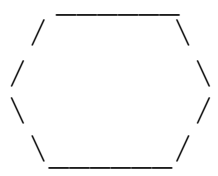
COSI 10A

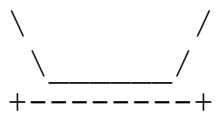


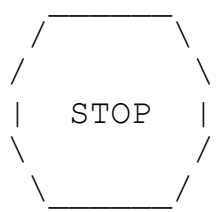
Functions

Review: Functions question

- Write a program to print these figures using functions

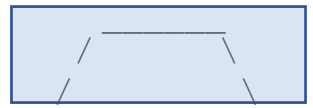






Review: version 3

Divide the code into functions

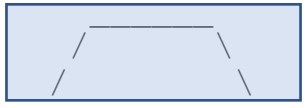


`egg_top`

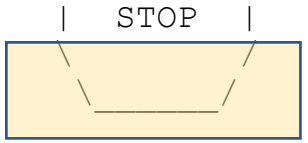
`egg_bottom`

`egg_bottom`

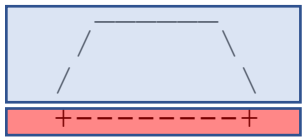
`line`



`egg_top`



`egg_bottom`



`egg_top`

`line`

- `egg_top`: reused on stop sign, hat
- `egg_bottom`: reused on teacup, stop sign
- `line`: used on teacup, hat

Review: version 3

```
# Prints several figures, with methods for structure and redundancy.
def main():
    egg()
    tea_cup()
    stop_sign()
    hat()

# Draws the top half of an an egg figure.
def egg_top():
    print("  ______")
    print(" /      \\")
    print("/        \\")

# Draws the bottom half of an egg figure.
def egg_bottom():
    print("\\        /")
    print(" \\______/")

# Draws a complete egg figure.
def egg():
    egg_top()
    egg_bottom()
    print()
```

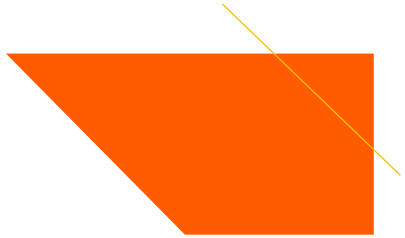
```
# Draws a teacup figure.
def tea_cup():
    egg_bottom()
    line()
    print()

# Draws a stop sign figure.
def stop_sign():
    egg_top()
    print("|  STOP  |")
    egg_bottom()
    print()

# Draws a figure that looks sort of like a hat.
def hat():
    egg_top()
    line()

# Draws a line of dashes.
def line():
    print("+--------+")
main()
```

Review: Why functions?

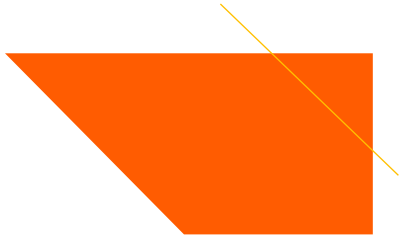
- Functions give you an opportunity to name a group of statements, which makes your program easier to read and debug
- Functions make a program smaller by eliminating repetitive code (or redundancy)
  - If you make a change you only have to make it in one place
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working program
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it

Class objectives

- Errors
- Data Types
- Variables



Debugging

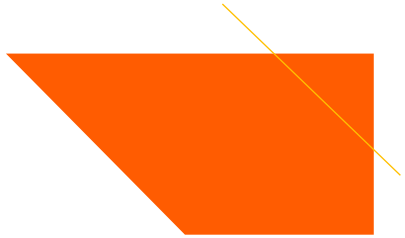


What is debugging?

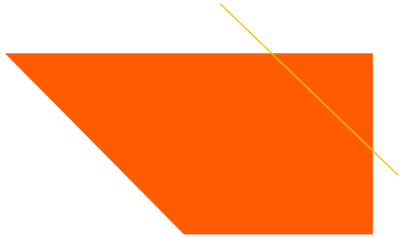
- Programming is error-prone
- The process of fixing errors in your programs is called debugging
- Three kinds of errors can occur in a program: **syntax errors**, **runtime errors**, and **semantic**

Errors

- **Syntax errors:** Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message
- **Runtime errors**: These errors do not appear until after the program has started running.
  - They usually indicate that something bad (or exceptional) has happened. They are rare in the simple programs.
- **Semantic(Logic) errors**: Your program will run successfully (no errors), but it will not do the right thing. It will do something else.
  - The meaning of the program (its semantics) is wrong. The problem is that the program you wrote is not the program you wanted to write.



Data and Expressions



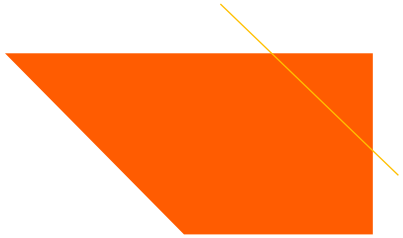
Data types

- Internally, computers store everything as 1s and 0s

```
104    01101000
'hi'   0110100001101001
'h'    01101000 (ASCII code)
```

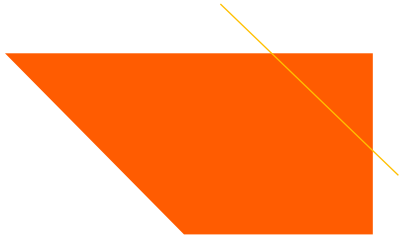
- How are h and 104 differentiated?

- **Type**: A category or set of data values
  - Constrains the operations that can be performed on data
  - Many languages ask the programmer to specify types
  - Examples: integer, real number, string



Python's number types

Name	Description	Examples
`int`	integers	`42, -3, 0, 926394`
`float`	real numbers	`3.1, -0.25`
`complex`		

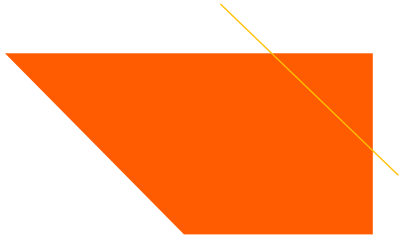


Expressions

- An **expression** is a value or operation that computes a value
- Examples:

```
1 + 4 * 5
(7 + 2) * 6 / 3
42.0
```

- The simplest expression is a literal value
- A complex expression can use operators and parentheses



Arithmetic operators

- An **operator** combines multiple values or expressions

`+`	addition
`-`	subtraction (or negation)
`*`	multiplication
`/`	division
`//`	integer division (a.k.a. leave off any remainder)
`%`	modulus (a.k.a. remainder)
`**`	exponent

- As a program runs, its expressions are evaluated

Integer division with //

- When we divide integers with //, the quotient is also an integer

  `14 // 4` is `3` not `3.5`

```
   3
4 ) 14
    12
     2
```

```
   4
10 ) 45
    40
     5
```

```
   52
27 ) 1425
    1350
      75
      54
      21
```

- More examples:

```
32 // 5       is 6
84 // 10      is 8
156 // 100    is 1
```

- Dividing by 0 causes an error when your program runs

Integer remainder with %

- The % operator computes the remainder from integer division

```
14 % 4          is 2
218 % 5         is 3
```

```
      3
  4 ) 14
      12
      2
```

```
     43
  5 ) 218
      20
      18
      15
      3
```

What is the result?

```
45 % 6
2 % 2
8 % 20
11 % 0
```

- Applications of % operator:
  - Obtain last digit of a number: `230857 % 10` is `7`
  - Obtain last 4 digits: `658236489 % 10000` is `6489`
  - See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`

Precedence

- **Precedence** is the order in which operators are evaluated
- Generally operators evaluate left-to-right

  `1 - 2 - 3` is `(1 - 2) - 3` which is `-4`
- But **`* / // %`** have a higher level of precedence than **`+ -`**

  `1 + 3 * 4` is `13`

  ```
  6 + 8 // 2 * 3
  6 +   4    * 3
  6 +      12
  ```
  is `18`
- Parentheses can force a certain order of evaluation `(1 + 3) * 4` is `16`
- Spacing does not affect order of evaluation `1+3 * 4-2` is `11`

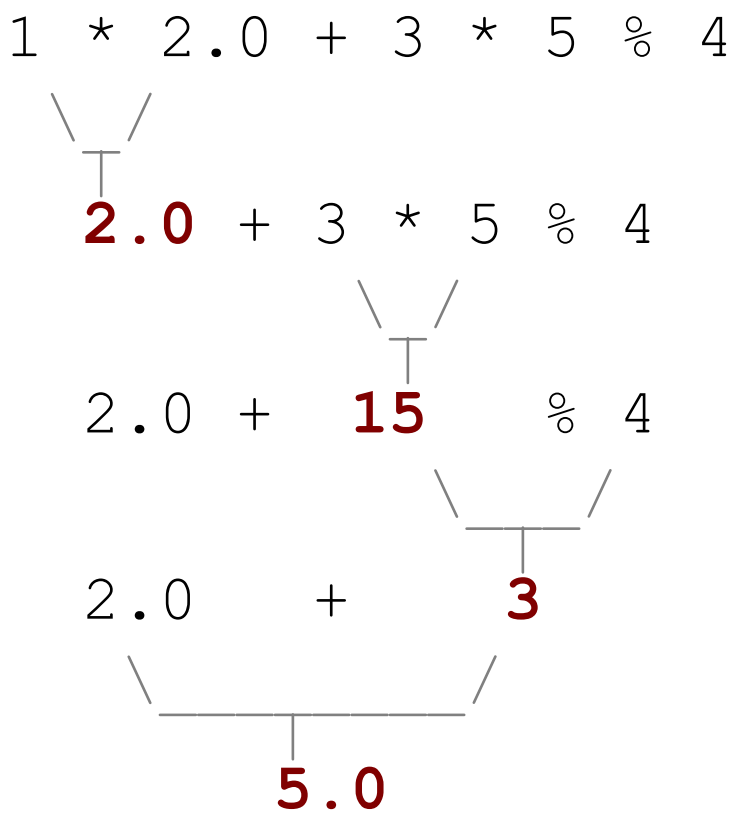
Order of operations

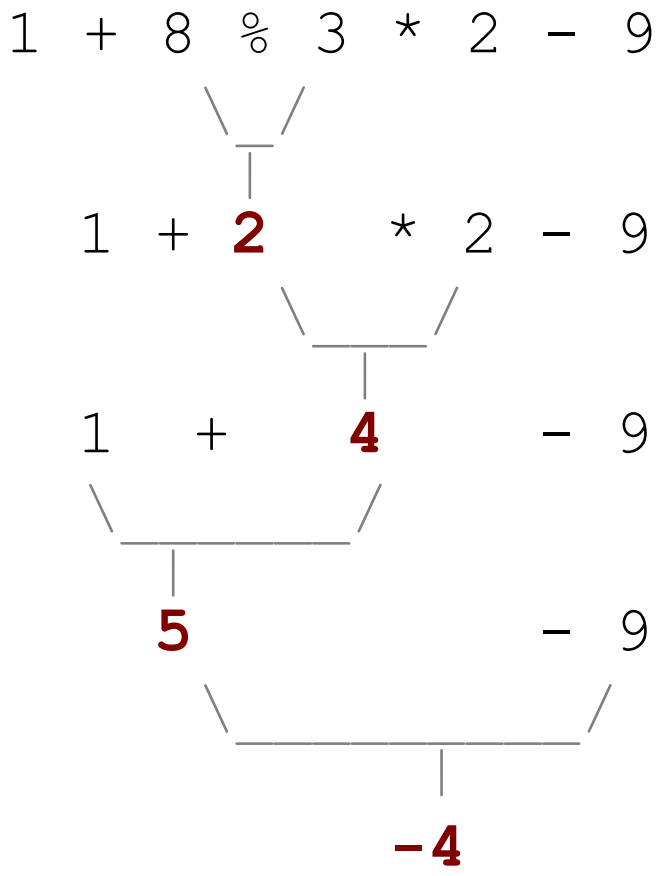
- When one operator appears in an expression, the order of evaluation depends on the **rules of precedence**

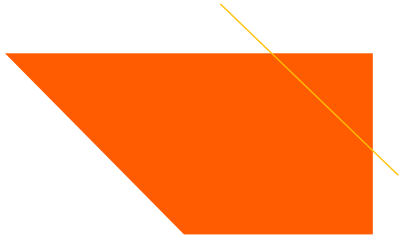
		Example	
( )	Parentheses	2 * (3 – 1) is 4	highest
**	Exponentiation	3 * 2**3 is 24	
*, /, //, %	Multiplication, Division, Modulus	7 // 2 * 4 is 12 7 // 3 % 3 is 2	
+, –	Addition, Subtraction	2 + 7 // 3 is 4	lowest

- Operators with the same precedence are evaluated from left to right (except exponentiation)

Precedence examples

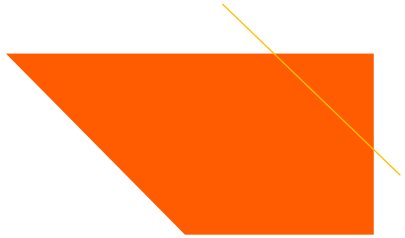






Precedence questions

- What values result from the following expressions?
  - `9 // 5`
  - `695 % 20`
  - `7 + 6 * 5`
  - `7 * 6 + 5`
  - `248 % 100 / 5`
  - `6 * 3 - 9 // 4`
  - `(5 - 7) * 2 ** 2`
  - `6 + (18 % (17 - 12))`

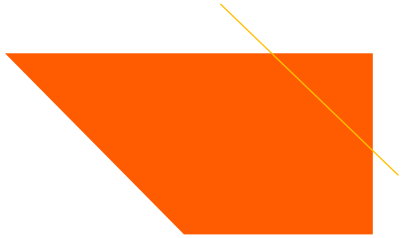


Variables

Example: receipt question

- What's bad about the following code?

```
# Calculate total owed, assuming 8% tax / 15% tip
print("Subtotal:")
print(38 + 40 + 30)

print("Tax:")
print((38 + 40 + 30) * .08)

print("Tip:")
print((38 + 40 + 30) * .15)

print("Total:")
print(38 + 40 + 30 + (38 + 40 + 30) * .15 + (38 + 40 + 30) * .08)
```



Example: receipt question

- What's bad about the following code?

```
# Calculate total owed, assuming 8% tax / 15% tip
print("Subtotal:")
print(38 + 40 + 30)

print("Tax:")
print((38 + 40 + 30) * .08)

print("Tip:")
print((38 + 40 + 30) * .15)

print("Total:")
print(38 + 40 + 30 + (38 + 40 + 30) * .15 + (38 + 40 + 30) * .08)
```

- The subtotal expression (38 + 40 + 30) is repeated
- So many print statements

Variables

- A **variable** is a piece of the computer's memory that is given a name and type, and can store a value
- Steps for using a variable:
  - **Declare/initialize it** - state its name and type and store a value into it
  - **Use it** - print it or use it as part of an expression

Declaration and Assignment

- **Variable declaration and assignment:** Sets aside memory for storing a value and stores a value into a variable
  - Variables **must be declared** before they can be used
  - The value can be an expression. The variable will store its result

- Syntax:

```
name = expression
```

- Example:

```
zipcode = 90210
myGPA = 1.0 + 2.25
```

zipcode	90210

myGPA	3.25

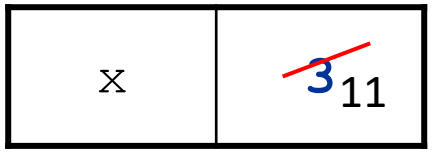
Using variables

- Once given a value, a variable can be used in expressions:

```
x = 3                  # x is 3
y = 5 * x - 1          # now y is 14
```

- You can assign a value more than once:

```
x = 3                  # 3 here

x = 4 + 7              # now x is 11
```



Assignment and algebra

- Assignment uses **=** , but it is not an algebraic equation.
  - **=** means, *"store the value at right in variable at left"*
  - The right side expression is evaluated first, and then its result is stored in the variable at left

- What happens here?

```
x = 3
x = x + 2    # ???
```