Expressions, Data Conversion, and Input

- Expressions
- Operators and Precedence
- Assignment Operators
- Data Conversion
- Input
- ***Reading*** for this class: *Dawson, Ch. 2*

Operators and Operands

- Operand: Can be any element that has some value:
  - A literal:

    `1`, `-2.5`, `True`, `False`, `"d"`, `"Hello World"`

  - A variable:

    `name`, `balance`, `course_title`

  - The result of a method call:

    `student.get_name()`

Operators and Operands

- Operator: Something that *computes a result* using one or more operands:

  1 + 2

  6 / 3

  not student_is_senior

  count += 1

  5 * 4 == 10 * 2

  18 – 6 != 6 - 18

Expressions

- An *expression* is a combination of one or more **operators** and **operands**
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Add	+	Integer (*floor*) Division	//
Subtract	-		
Multiply	*		
Divide	/		
Remainder	%	Exponent	**

- If either or both operands used by an arithmetic operator are floating point (i.e., **decimal**), then the result is a floating point

See `word_problems.py`

Division and Remainder

- The division operators (/ and //) work differently, depending on the types of operands supplied

14 / 3	equals	4.66666...
14 // 3	equals	4
8 / 12	equals	0.666666...
8 // 12	equals	0

- Try out the following and see what they do:

```
4 / 3    4.0 / 3    4 // 3    4.0 // 3
```

- The remainder operator (%) returns the remainder after dividing the second operand into the first

14 % 3	equals	2
8 % 12	equals	8

Operator Precedence

- Operands and operators can be combined into **complex expressions**

```
result  =  total + count / maxi - offset
```

- Operators have a well-defined **precedence** which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but **parentheses** can be used to **force the evaluation order**
- **See link for precedence information:**

http://www.tutorialspoint.com/python/python_basic_operators.htm

Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e
  1   2   3   4
```

```
a + b * c - d / e
  3   1   4   2
```

Without parentheses:

```
a / b + c - d % e
  1   3   4   2
```

With parentheses:

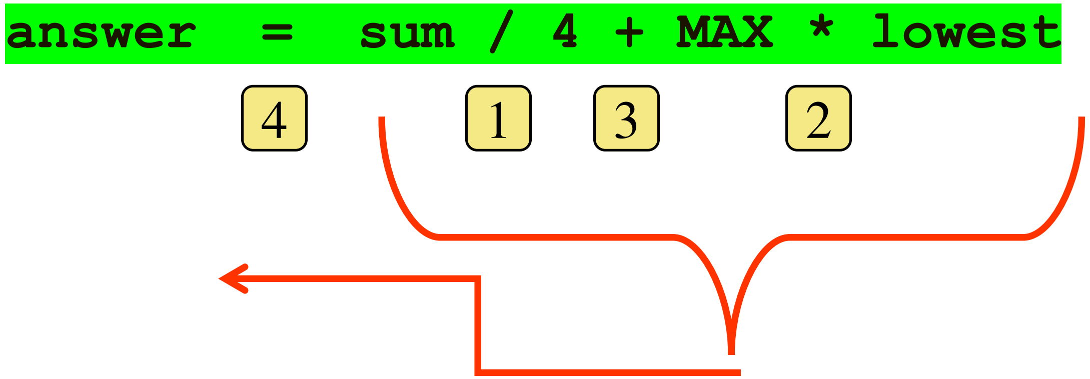
```
a / (b + c) - d % e
  2    1    4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

Assignment Revisited

- The assignment operator has a **lower** precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

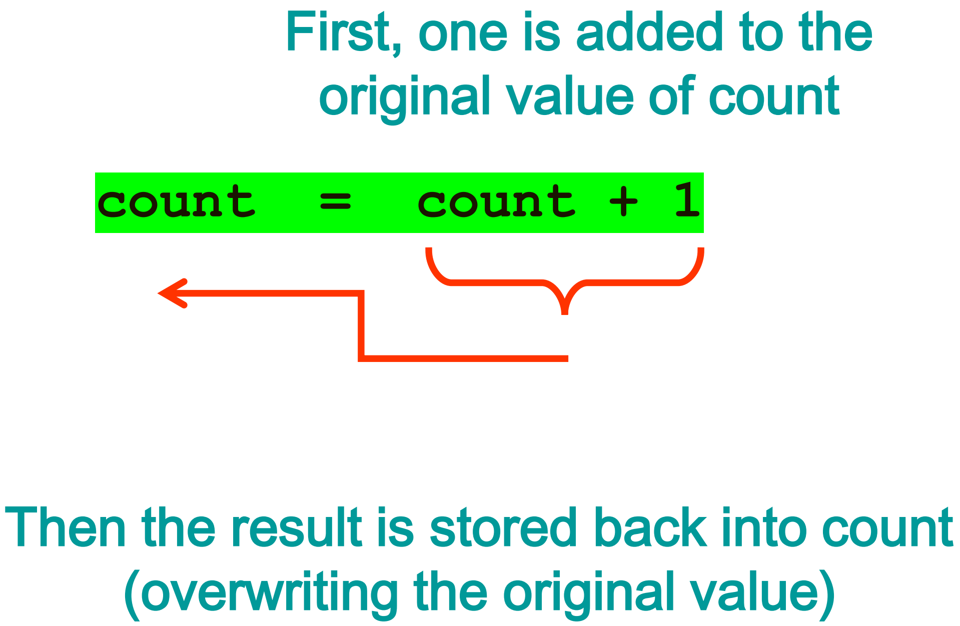


```
answer  =  sum / 4 + MAX * lowest
```

Then the result is stored in the variable on the left hand side

Assignment Revisited

- The right and left hand sides of an assignment statement can contain the same variable



First, one is added to the original value of count

```
count  =  count + 1
```

Then the result is stored back into count (overwriting the original value)

Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- Python provides *assignment operators* to simplify that process
- For example, the statement

```
num += count
```

is equivalent to

```
num = num + count
```

Assignment Operators

- There are many assignment operators in Python, including the following:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Assignment Operators

- The right hand side of an assignment operator can be a complex expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
                3        1     2
```

Expressions such as the former, if used correctly, can enhance your code's readability

Assignment Operators

- The behavior of some assignment operators depends on the types of the operands

- If the operands to the **+=** operator are strings, the assignment operator performs string concatenation

- The behavior of an assignment operator (**+=**) is always consistent with the behavior of the corresponding operator (**+**)

Data Conversion

- Sometimes it is convenient to convert data from one **type** to another
- For example, in a particular situation we may want to treat an integer as a decimal value
- These conversions **do not change** the type of a variable or the value that's stored in it – they only convert **the value itself** as part of a computation

Data Conversion

- Conversions must be handled carefully to avoid losing information
- ***Widening conversions*** are safest because they tend to go from a less precise data type to a more precise one (such as an **`int`** to a **`float`**)
- ***Narrowing conversions*** can lose information because they go from a more precise data type to a less precise one (such as a **`float`** to an **`int`**)
- Other types of data conversions involve changing to a completely different form, such as converting a type to or from a string

Method Conversion

- The conversions you see at this stage will involve the use of methods:

```
str (value)
```

```
int (value)
```

```
float (value)
```

- Replace **value** with what you wish to convert
- For example:

```
x = 1.8
```

```
y = 10
```

```
print (int (x)) →  1
```

```
print (float(y)) → 10.0
```

Character Arithmetic

- Because characters are associated with 16-bit integer values, you can do *arithmetic with characters*!

- For example, the expression

```
ord('b') - ord('a')
```

- will evaluate to `1` because the integer value of `'b'` is one more than that of `'a'`

- As such, you may find it useful to become more comfortable at converting back and forth between characters and their integer equivalents

Character Arithmetic

• Statements:	Prints:
`print('a')`	`a`
`print(97)`	`97`
`print(ord('a'))`	`97`
`print(chr(97))`	`a`

- **These lines will . . . . print as:**

```
i = 0
print (chr(ord('A') + i))
i += 1
print (chr(ord('A') + i))
i += 1
print (chr(ord('A') + i))
```

```
A
B
C
```

Character Arithmetic

- **Why does...**	**print as?**
`print('a')`	`a`
	Literal: **'a'**
`print(97)`	`97`
	Literal: **97**
`print(ord ('a'))`	`97`
	Character value **converted** to an `int` value: **97**
`print(chr (97))`	`a`
	Integer value **converted** to a `char` value: **'a'**

Character Arithmetic

- **Why does...**

```
i = 0
print (chr(ord('A') + i))
i += 1
print (chr(ord('A') + i))
i += 1
print (chr(ord('A') + i))
```

print as?

```
A
B
C
```

(It has to do with the **steps of conversion**...)

1) 'A' → **value** of 'A' is converted to **int:** 97

2) 97 + ***i*** → evaluates to an **int:** 98

3) **98** is converted to a character, which gets printed.

(*NOTE: The letters are printed successively because* ***i*** *starts off as zero and gets incremented*)

Reading Input

- Programs generally need input on which to operate
- The `input` method allows us to get this information from the user, when writing a command-line application
- It can also be used to halt program execution until the user presses **`Enter`**
- To use it, you will need:

1) The method name: **`input`**

2) Prompt text

Reading Input

- The input method will:

  1) Print your specified prompt text

  2) Wait for the user to press **`Enter`**

  3) Return the user's input in the form of a *string object* (an empty string, if the user entered no text)

- To halt program execution, you can use input *without storing the result.*
- This can be useful when you want the program to *stop* at certain points

Reading Input

- Examples:

```
name = input ("Name: ")
age = int (input ("Age: "))
height = float (input ("Height (m): ")
input ("Press Enter to continue")
print ("Your name is," name)
print ("You are", age, "years old")
print ("You are", height, "meters tall")
```

See:

input_demo.py   trust_fund_bad.py

personal_greeter.py   trust_fund_good.py

Interactive Applications (CLI)

- An interactive program with a command line interface contains a sequence of steps to:
  - Prompt the user
  - Get the user’s responses
  - Process the data as input is received (or after)

```
name = input("Enter name: ")
age = int( input("Enter age: "))
money = float( input("Money: $"))
```

See `useless_trivia.py`

The *math* module

- The `math` module is part of the Python standard library. To use it, we must first have the following line at the start of our program:

```
import math
```

- The `math` module contains methods that perform various mathematical functions
- These include:
  - –square root
  - –exponentiation
  - –logarithms
  - –trigonometric functions

See `using_math.py`

`https://docs.python.org/3.4/library/math.html`

The *math* Module

- In addition, Python also has several built-in methods that support mathematical operations, such as **`abs`** (for absolute value) and **`min`** and **`max`** (for the minimum or maximum of a list of values)
- Examples of use:

```
value = math.cos(90) + math.sqrt(delta)

print(abs(value))

print (math.log2 (16.0)) ==> 4.0

print (min (2, 4)) ==> 2

print (max (1, 5)) ==> 5
```

The *random* module

- The `random` module is for introducing elements of randomness
- It must be imported:

```
import random
```

- Gives methods such as:

`randint(a, b)` **: a <= x <= b**

`random()` **: 0.0 <= x < 1.0 (float type)**

`choice(seq)` **: some random element from a sequence**

The *random* module

- More random methods:

https://docs.python.org/3.4/library/random.html

- Put the code below into a file and run it. Also, make up some of your own and experiment:

```
import random
print (random.random())
print (random.randint(1, 10))
print (random.randint(20, 200))
```

Interactive Applications (CLI)

- Consider `quadratic.py`

```
# We will not need this right away, but
# eventually, we will...
import math

# First, get A, B, and C from user
a = float (input ("Enter the coefficient of x
  squared: "))
b = float (input ("Enter the coefficient of x: "))
c = float (input ("Enter the constant: "))
```

We have the input values, now what?

- To solve the quadratic equation, we need to program in Python the formulas learned in high school algebra:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- How do we program those equations?
- We need to use
  - –The *`math`* module,
  - –Expression Evaluation, and
  - –Assignment

*FYI, this value is called the **discriminant**!* → $b^2 - 4ac$

Solving Quadratic Equations

```
disc = b*b - 4*a*c
root1 = ((-1 * b) + math.sqrt(disc)) / (2 * a)
root2 = ((-1 * b) - math.sqrt(disc)) / (2 * a)
```

- However, this program to solve for the roots of a quadratic equation is **deficient**!
- The equations for calculating the roots are correct but are not used correctly in the program
- It only gives correct answers so long as the coefficients entered actually belong to a **quadratic** equation with **real** roots

Solving Quadratic Equations

- User can enter any values for “a”, “b”, and “c”, which can create special cases that the formula cannot accommodate
- Let’s try **`a = 2`** , **`b = 3`** , and **`c = 4`** (demo)
- What happens?
- **Answer**: A negative discriminant, which has no real square root

```
discriminant = 3 * 3 – 4 * 2 * 4
discriminant =  9 – 32
```

`discriminant = -23`

The ***`math.sqrt`*** method cannot handle this!

Solving Quadratic Equations

- However, there is **the “imaginary” number i** (the square root of -1)

In math: $\sqrt{-7} \Rightarrow i * \sqrt{7}$

String: `“i * “ + str(math.sqrt(7)) => “i * 2.6457513110645907"`

Equation may have **complex** roots (e.g., $5 + i\sqrt{7}$ and $5 - i\sqrt{7}$)

- How do we accommodate such user input?
- **Answer:** check discriminant value:
  - *Positive*: Use given formula
  - *Negative*: Construct complex root strings
  - *Zero*: `-b/(2a)` (Need not print value twice!)

Solving Quadratic Equations

- Other possible problems:
  - *a = **0** (but not b)*: Formula divides by 2 * a, leading to an error if a equals **0**. (Equation is **linear, not quadratic**, so the only root is the **y-intercept**)
  - *a and b (but not c) are **0***: A horizontal line that **never touches the x-axis**, so no roots
  - *All three are **0***: The x-axis itself, so all values are roots (in the sense that **any** value of x would satisfy `0*x^2 + 0*x + 0 = 0`
- Our program must account for all these possibilities – by **making decisions!**

Control Flow

- Up until now, each program has been a linear sequence of steps
- First statement, second, and so forth…in sequence
- To make decisions while solving a quadratic equation, we need to direct the program to different statements based upon contingencies of user input
- We will see how to do that shortly