CptS 111, Spring 2023
Lect. #3, Jan. 18, 2023
Class Notes

Today's Agenda:

1. Augmented assignment, math operators, and math precedence
2. A little more on `print()`
3. Division vs floor division, modulo function, and `divmod()`
4. Modules

Ch. 2 (cont.)

1. Augmented Assignment (Compound Operators), Math Operators, and Math Precedence

A. Augmented Assignment and Math Operators
When programming, we can use shorthand to perform some operations. Consider the following:

```
i = 0
i = i + 1
```

It's often convenient to use augmented assignment:

```
i = 0
i += 1
```

I know this looks odd, but the end result is the same (and you get used to it). The second statement is shorthand for `i = i + 1`. We can use other operations as well. In fact, we can use any of the basic math operators:

```
+:  addition
-:  substraction
*:  multiplication
/:  division
**: exponentiation
```

Let's consider some examples.

In [1]:
```
# Augmented assignment using +

x = 0
x += 2
print(x)
```

```
2
```

In [2]:
```
# Augmented assignment using -

x -= 1
print(x)
```

```
1
```

In [3]:
```
# Augmented assignment using * (multiplication)

x *= 15
print(x)
```

```
15
```

In [4]:
```
# Augmented assignment using /; notice that / gives a float!

x /= 3
print(x)
```

```
5.0
```

B. Math Precedence
When writing arithmetic expressions, we have to keep in mind the rules of math precedence (see Table 2.4.2), i.e., which operations are performed first in an expression. Consider the following:

```
x = 3 + 4 * 8 / 3 - 5 + 3 ** 2
```

If we don't use parentheses to indicate exactly how to evaluate the expression on the right, Python will first do the exponentiation (i.e., `3 ** 2`, which is `3` squared) because exponentiation has the highest precedence in the expression. Then it will perform multiplication and division from left to right. Multiplication `*` and division `/` have equal precedence. Finally, Python will perform addition and subtraction from left to right. Addition `+` and subtraction `-` have equal precedence. Using parentheses to demonstrate this, we have:

```
x = ((3 + ((4 * 8) / 3)) - 5)) + (3 ** 2)
```

We can see that this is true:

In [5]:
```
# Default precedence for math operators

3 + 4 * 8 / 3 - 5 + 3 ** 2
```

Out[5]: 17.666666666666664

In [6]:
```
# "Proof" of default precedence

((3 + ((4 * 8) / 3) - 5)) + (3 ** 2)
```

Out[6]: 17.666666666666664

If we want to perform operations in a different order, we can always use parentheses. For example:

In [7]:
```
# Changing the default order of precedence

(3 + 4) * 8 / 3 - (5 + 3 ** 2)
```

Out[7]: 4.666666666666668

2. A Little More on `print()`

We enclose text we want to print in either single or double quotes.

In [8]:
```
# Use of single quotes

print('Hello, World!')
```

```
Hello, World!
```

The choice of single quotes or double quotes is a matter of preference, but if there's a single quote in the text to be printed, we need to use double quotes or we'll get an error message (throw an exception). Alternatively, we can use the escape character `\`.

In [9]:
```
# But we can't use three single quotes!

print('didn't')
```

```
  Input In [9]
    print('didn't')
                 ^
SyntaxError: unterminated string literal (detected at line 3)
```

In [10]:
```
# Instead, use double quotes to delimit the string

print("didn't")
```

```
didn't
```

In [11]:
```
# Alternatively, use the escape character which tells Python to
# print the character following it.

print('didn\'t')
```

```
didn't
```

When we print a variable or literal, a float, or an integer, we don't have to use quotes.

In [12]:
```
# cat is variable, pi is float, 42 is integer

cat = 'dog'
print(cat, 3.141592653, 42)
```

```
dog 3.141592653 42
```

What if we just want to print the first two decimal places of 3.141592653? We can use string formatting in our print command. We'll discuss this more in Ch. 3, but for now just remember that `{:.2f}` means a float with 2 decimal places. Similarly, `{:.4f}` means a float with 4 decimal places.

In [13]:
```
# Use f-string string formatting for 2 decimal places

pi = 3.141592653
print(f'pi to 2 decimal places is {pi:.2f}.')
```

```
pi to 2 decimal places is 3.14.
```

In [14]:
```
# Or 4 decimal places; note how rounding occurs!

print(f'pi to 4 decimal places is {pi:.4f}.')
```

```
pi to 4 decimal places is 3.1416.
```

3. Division, Floor Division, the Modulo Function, and the `divmod()` Function

Note the following:

1. Division (/) *ALWAYS* results in a float regardless of operand types.
2. Both floor division (//) and modulo (`%`) *result in an integer if the operands are integers* and *in a float if one (or both) of the operands is a float*.

Floor division and the modulo function are surprisingly useful.

- floor division //: cuts off the remainder, i.e., it rounds down to a whole number
- modulo function %: gives the remainder

Consider the following:

In [15]:
```
# Division with integers

4 / 2
```

Out[15]: 2.0

In [16]:
```
# Division with a float

4 / 2.0
```

Out[16]: 2.0

In [17]:
```
# Floor division with integers

13 // 5
```

Out[17]: 2

In [ ]:
```
# Floor division with a float

13 // 5.0
```

In [18]:
```
# Modulo with an integer

13 % 5
```

Out[18]: 3

In [19]:
```
# Modulo with a float

13 % 5.0
```

Out[19]: 3.0

In [20]:
```
# Using simultaneous assignment, i.e., n values on left and n on right

x, y = 13 // 5, 13 % 5
print(x, y)
```

```
2 3
```

Notice two things about the last operations:

1. We can use two lvalues on the left if we have two operations or values to the right of the assignment operator. Lvalues and operations or values are both separated by commas. This overall operation is called simultaneous assignment.
2. `x` gives us the whole number (rounding down) and `y` gives us the remainder.

There's actually a function that performs both operations and gives two results. It's called the `divmod()` function. **You need to learn this function!** Consider:

In [21]:
```
# divmod(13, 5) does the same thing as 13 // 5, 13 % 5

x, y = divmod(13, 5) # Same as 13 // 5, 13 % 5
print(x, y)
```

```
2 3
```

We get the same result! But how is the `divmod()` function useful? Consider zCA 2.6.2 (Compute change).

In [22]:
```
# zyBook challenge activity (zCA) 2.6.2 using // and % (19)

amt_to_change = int(input())
num_fives = amt_to_change // 5
num_ones = amt_to_change % 5
print('Change for $', amt_to_change, ':', sep='')
print(num_fives, 'five-dollar bill(s) and', num_ones, 'one-dollar bill(s)')
```

```
19
Change for $19:
3 five-dollar bill(s) and 4 one-dollar bill(s)
```

In [23]:
```
# Another solution using divmod()

amt_to_change = int(input())
num_fives, num_ones = divmod(amt_to_change, 5)
print('Change for $', amt_to_change, ':', sep='')
print(num_fives, 'five-dollar bill(s) and', num_ones, 'one-dollar bill(s)')
```

```
19
Change for $19:
3 five-dollar bill(s) and 4 one-dollar bill(s)
```

Using the `divmod()` function in Python saves us some typing, and we'll use this function quite a bit in CptS 111! Let's consider another zCA--2.5.2.

In [24]:
```
# Calculate the volume of a sphere for some radius.  Recall, sphere_volume
# is (4/3)*pi*r^3.

pi = 3.14159
sphere_radius = float(input())

sphere_volume = (4 / 3) * pi * (sphere_radius ** 3)
print(f'Sphere volume: {sphere_volume:.2f}')  # Using f-string string formatting
```

```
1
Sphere volume: 4.19
```

4. Modules

We've learned some basic functions in Python, e.g., `print()`, `input()`, `type()`, `int()`, `float()`, and `divmod()`. These functions are known as *built-in* functions because they come "preloaded" in Python. However, there are a relatively small number of these basic built-in functions. In fact, there are only 72:

```
'abs','all','any','ascii','bin','bool','breakpoint','bytearray',
'bytes','callable','chr','classmethod','compile','complex',
'copyright','credits','delattr','dict','dir','display','divmod',
'enumerate','eval','exec','filter','float','format','frozenset',
'get_ipython','getattr','globals','hasattr','hash','help','hex',
'id','input','int','isinstance','issubclass','iter','len',
'license','list','locals','map','max','memoryview','min','next',
'object','oct','open','ord','pow','print','property','range',
'repr','reversed','round','set','setattr','slice','sorted',
'staticmethod','str','sum','super','tuple','type','vars','zip'
```

Rather than having a huge number of functions in Python, we import modules when we want to use specialized functions. A **module** is just code that's stored in a .py file for use in another module or in a Python **script**, i.e., a program we've written in, e.g., an IDLE Editor window. Each module contains functions designed for a specific purpose.

Python comes with a set of standard modules that is quite extensive, but in addition, there are thousands and thousands of modules available that have been written by both professionals and enthusiasts. Some of the standard modules included with Python are:

```
math - math functions
cmath - complex math functions
statistics - statistics functions
random - random numbers
turtle - graphics
os - operating system
time - time access and conversions
mailbox - mailbox manipulation
calendar - calendar functions
```

We'll work with a number of these in future chapters, but for now we'll just consider the `math` module.

In order to use a module, we have to import it. We'll learn two different ways of importing modules now and will learn several more in Ch. 7.

A. Basic Import Statement
The simplest import statement requires the use of **dot notation** by which we mean that *when we use a function in the module, the name of the module must be given, followed by a period, and then by the name of the function*.

```
import <module>
```

Let's give this a try.

In [25]:
```python
# Basic import statement

import math
print('pi =', math.pi)
print('5! =', math.factorial(5))
print('sqrt(25) =', math.sqrt(25)) # sqrt() always returns a float
```

```
pi = 3.141592653589793
5! = 120
sqrt(25) = 5.0
```

B. Import Statement with Alias
Sometimes we want to use dot notation so we know that a function has been imported, but we want to shorten the module name if we're going to use a lot of its functions to decrease the amount of typing required. This can be done very simply.

```
import <module> as <name>
```

In [26]:
```python
# Import statement with an alias

import math as m
print('pi =', m.pi)
print('5! =', m.factorial(5))
print('sqrt(25) =', m.sqrt(25))
```

```
pi = 3.141592653589793
5! = 120
sqrt(25) = 5.0
```