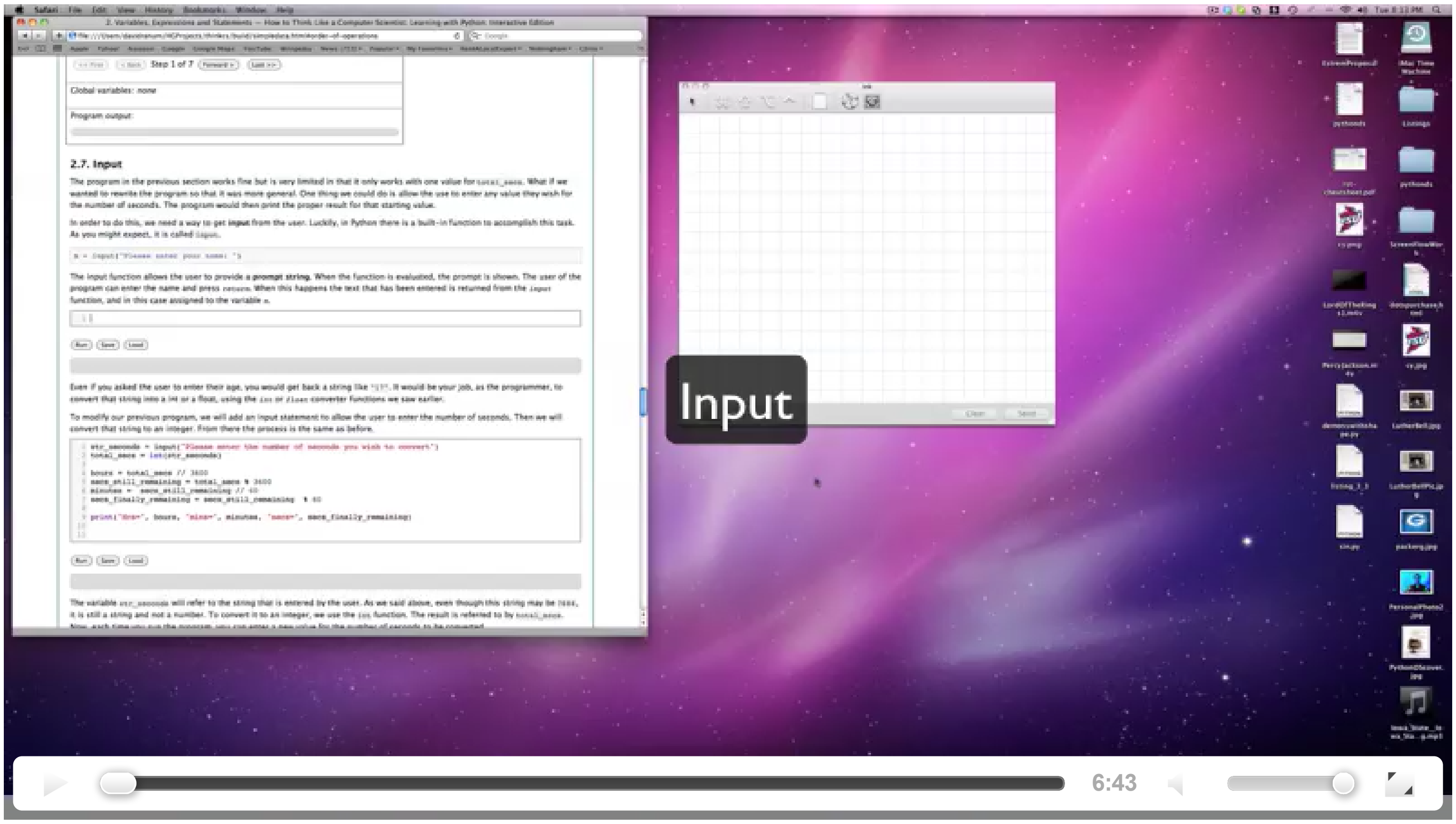
Input



The program in the previous section works fine but is very limited in that it only works with one value for `total_secs`. What if we wanted to rewrite the program so that it was more general. One thing we could do is allow the use to enter any value they wish for the number of seconds. The program could then print the proper result for that starting value.

In order to do this, we need a way to get **input** from the user. Luckily, in Python there is a built-in function to accomplish this task. As you might expect, it is called `input`.

```
n = input("Please enter your name: ")
```

The input function allows the user to provide a **prompt string**. When the function is evaluated, the prompt is shown. The user of the program can enter the name and press return. When this happens the text that has been entered is returned from the input function, and in this case assigned to the variable n. Make sure you run this example a number of times and try some different names in the input box that appears.

```
n = input("Please enter your name: ")
print("Hello", n)

```

ActiveCode: 1 (inputfun)

Run

```
Hello ji
```

It is very important to note that the `input` function returns a string value. Even if you asked the user to enter their age, you would get back a string like `"17"`. It would be your job, as the programmer, to convert that string into an int or a float, using the `int` or `float` converter functions we saw earlier.

To modify our previous program, we will add an input statement to allow the user to enter the number of seconds. Then we will convert that string to an integer. From there the process is the same as before. To complete the example, we will print some appropriate output.

```
str_seconds = input("Please enter the number of seconds you wish to convert")
total_secs = int(str_seconds)

hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes =  secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining  % 60

print("Hrs=", hours, "mins=", minutes, "secs=", secs_finally_remaining)

```

ActiveCode: 2 (int_secs)

Run

```
Hrs= 0 mins= 5 secs= 0
```

The variable `str_seconds` will refer to the string that is entered by the user. As we said above, even though this string may be `7684`, it is still a string and not a number. To convert it to an integer, we use the `int` function. The result is referred to by `total_secs`. Now, each time you run the program, you can enter a new value for the number of seconds to be converted.

Check your understanding

sdat-9: What is printed when the following statements execute?

```
n = input("Please enter your age: ")
# user types in 18
print ( type(n) )
```

[✓ a) <class 'str'>]

b) <class 'int'>

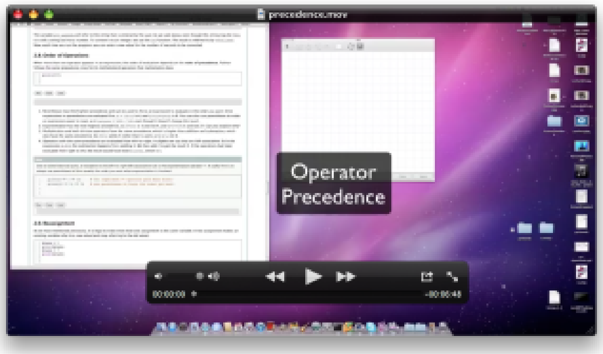
c) <class 18>

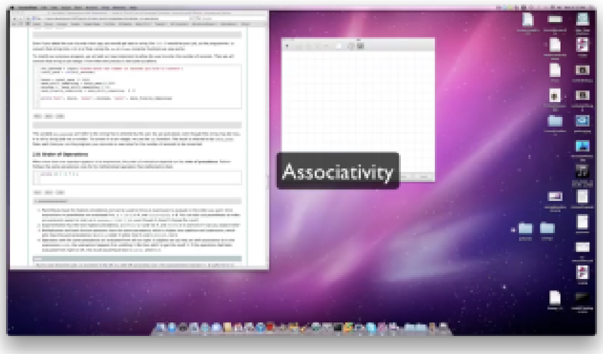
d) 18

Check Me Compare Me

Correct!! All input from users is read in as a string.

Order of Operations





When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.
2. Exponentiation has the next highest precedence, so `2**1+1` is 3 and not 4, and `3*1**3` is 3 and not 27. Can you explain why?
3. Multiplication and both division operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So `2*3-1` yields 5 rather than 4, and `5-2*2` is 1, not 6.
4. Operators with the *same* precedence are evaluated from left-to-right. In algebra we say they are *left-associative*. So in the expression `6-3+2`, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been `6-(3+2)`, which is 1.

Note

Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator **. A useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

```
print(2 ** 3 ** 2)     # the right-most ** operator gets done first!
print((2 ** 3) ** 2)   # use parentheses to force the order you want!

```

ActiveCode: 3 (ch02_23)

Run

```
512
64
```

Check your understanding

sdat-10: What is the value of the following expression:

```
16 - 2 * 5 // 3 + 1
```

[✓ a) 14]

b) 24

c) 3

d) 13.667

Check Me Compare Me

Correct!! Using parentheses, the expression is evaluated as (2*5) first, then (10 // 3), then (16-3), and then (13+1).

sdat-11: What is the value of the following expression:

```
2 ** 2 ** 3 * 3
```

[✓ a) 768]

b) 128

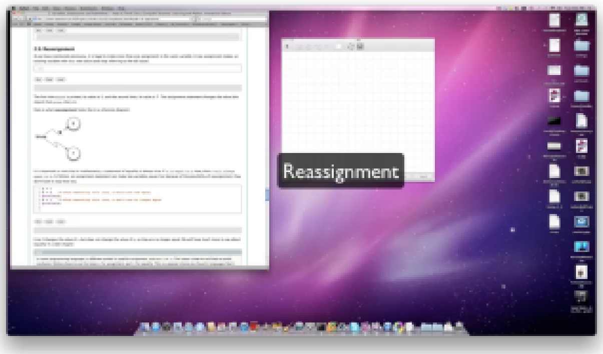
c) 12

d) 256

Check Me Compare Me

Correct!! Exponentiation has precedence over multiplication, but its precedence goes from right to left! So 2 ** 3 is 8, 2 ** 8 is 256 and 256 * 3 is 768.

Reassignment



As we have mentioned previously, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
bruce = 5
print(bruce)
bruce = 7
print(bruce)

```

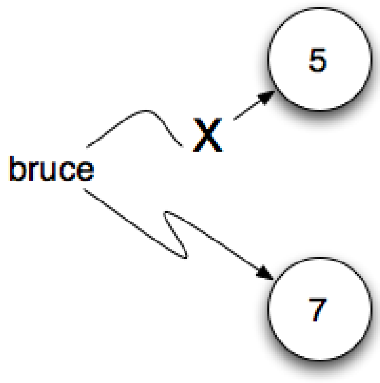
ActiveCode: 4 (ch07_reassign1)

Run

```
5
7
```

The first time `bruce` is printed, its value is 5, and the second time, its value is 7. The assignment statement changes the value (the object) that `bruce` refers to.

Here is what **reassignment** looks like in a reference diagram:



It is important to note that in mathematics, a statement of equality is always true. If `a is equal to b` now, then `a will always equal to b`. In Python, an assignment statement can make two variables refer to the same object and therefore have the same value. They appear to be equal. However, because of the possibility of reassignment, they don't have to stay that way:

```
1 a = 5
2 b = a    # after executing this line, a and b are now equal
3 print(a,b)
4 a = 3    # after executing this line, a and b are no longer equal
5 print(a,b)
6
```

ActiveCode: 5 (ch07_reassign2)

Run

```
5 5
3 5
```

Line 4 changes the value of `a` but does not change the value of `b`, so they are no longer equal. We will have much more to say about equality in a later chapter.

Note

In some programming languages, a different symbol is used for assignment, such as `<-` or `:=`. The intent is that this will help to avoid confusion. Python chose to use the tokens `=` for assignment, and `==` for equality. This is a popular choice also found in languages like C, C++, Java, and C#.

Check your understanding

sdat-12: After the following statements, what are the values of x and y?

```
x = 15
y = x
x = 22
```

a) x is 15 and y is 15

b) x is 22 and y is 22

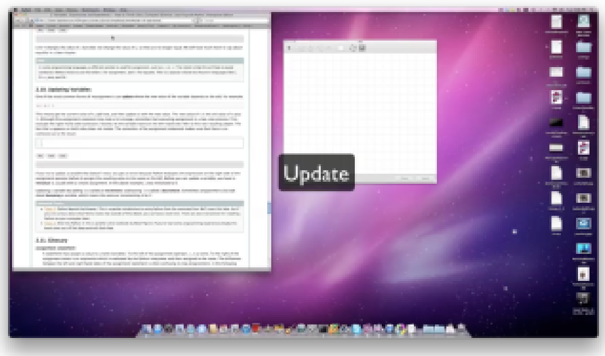
c) x is 15 and y is 22

[✓ d) x is 22 and y is 15]

Check Me Compare Me

Correct!! Yes, x has the value 22 and y the value 15.

Updating Variables



One of the most common forms of reassignment is an **update** where the new value of the variable depends on the old. For example,

```
x = x + 1
```

This means get the current value of x, add one, and then update x with the new value. The new value of x is the old value of x plus 1. Although this assignment statement may look a bit strange, remember that executing assignment is a two-step process. First, evaluate the right-hand side expression. Second, let the variable name on the left-hand side refer to this new resulting object. The fact that `x` appears on both sides does not matter. The semantics of the assignment statement makes sure that there is no confusion as to the result.

```
x = 6        # initialize x
print(x)
x = x + 1    # update x
print(x)

```

ActiveCode: 6 (ch07_update1)

Run

```
6
7
```

If you try to update a variable that doesn’t exist, you get an error because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left. Before you can update a variable, you have to **initialize** it, usually with a simple assignment. In the above example, `x` was initialized to 6.

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**. Sometimes programmers also talk about **bumping** a variable, which means the same as incrementing it by 1.

Check your understanding

sdat-13: What is printed when the following statements execute?

```
x = 12
x = x - 1
print (x)
```

- a) 12
- b) -1
- [✓ c) 11]
- d) Nothing. An error occurs because x can never be equal to x - 1.

Check Me   Compare Me

Correct!! Yes, this statement sets the value of x equal to the current value minus 1.

sdat-14: What is printed when the following statements execute?

```
x = 12
x = x - 3
x = x + 5
x = x + 1
print (x)
```

- a) 12
- b) 9
- [✓ c) 15]
- d) Nothing. An error occurs because x cannot be used that many times in assignment statements.

Check Me   Compare Me

Correct!! Yes, starting with 12, subtract 3, than add 5, and finally add 1.

sdat-15: Construct the code that will result in the value 134 being printed.

Drag from here

Drop blocks here

```
mybankbalance = 100
```

```
mybankbalance = mybankbalance + 34
```

```
print(mybankbalance)
```

Check Me Reset

Perfect!

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```


```

ActiveCode: 7 (scratch_02)

Run