Week 3

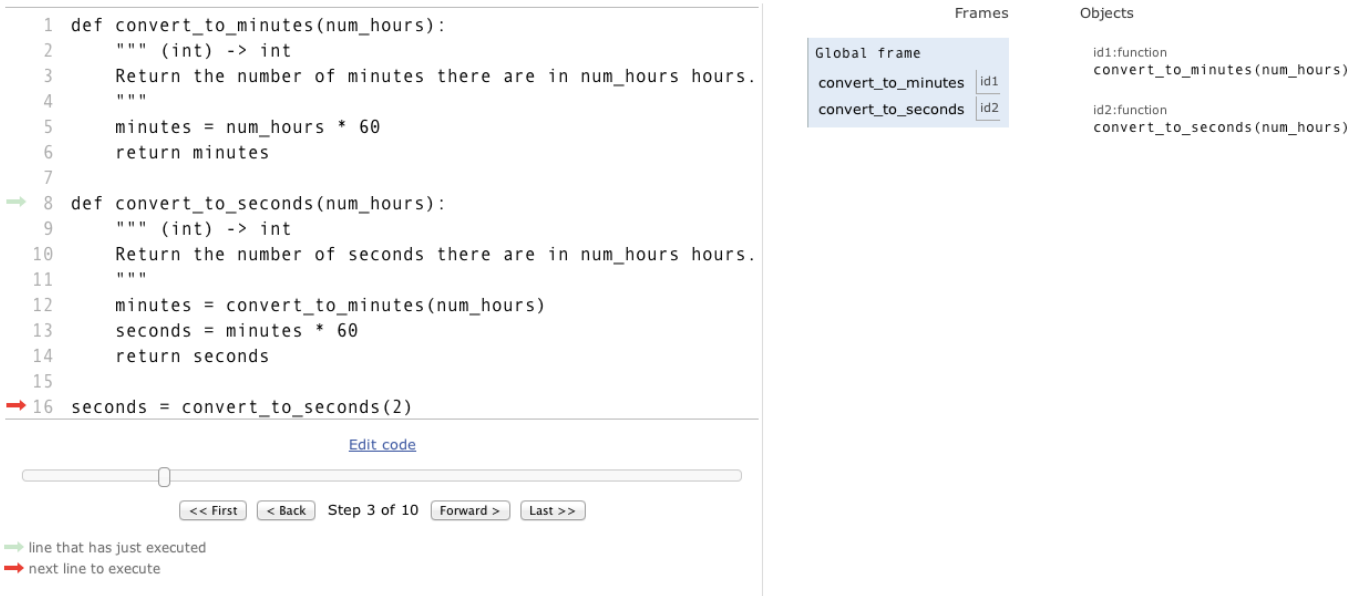
Video 1: Functions, Variables, and the Call Stack

Understanding Scope

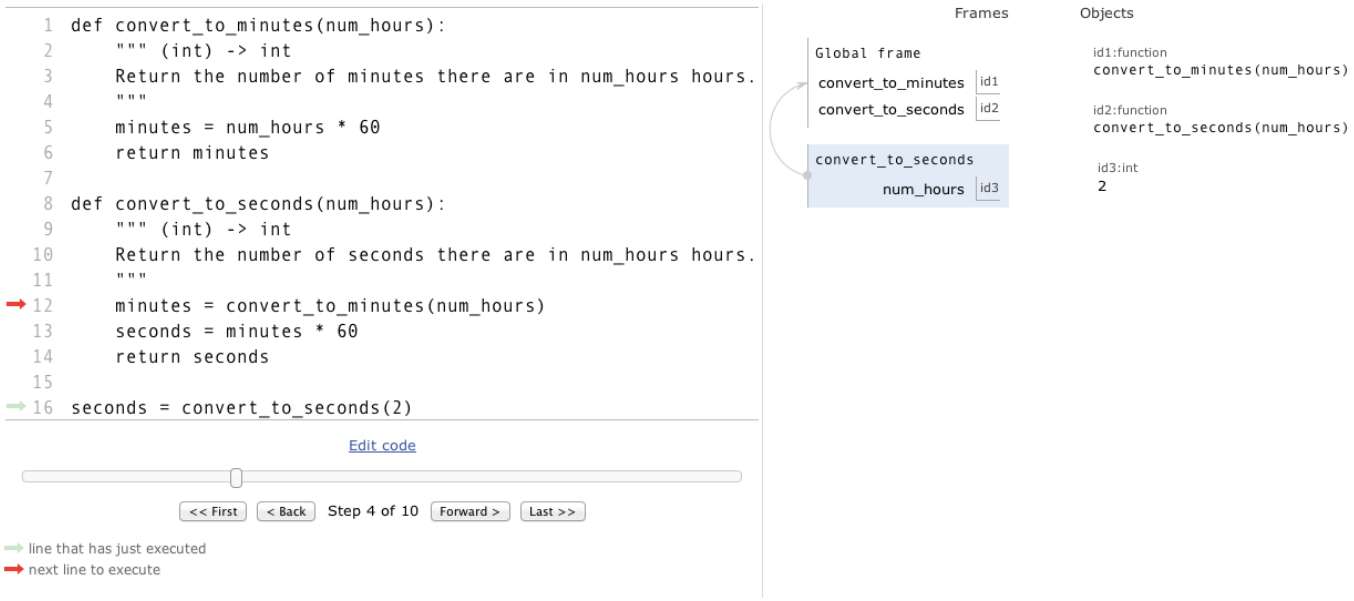
Below is an explanation and review of the example used in the video.

```
def convert_to_minutes(num_hours):
    """ (int) -> int
    Return the number of minutes there are in num_hours hours.
    """
    minutes = num_hours * 60
    return minutes

def convert_to_seconds(num_hours):
    """ (int) -> int
    Return the number of seconds there are in num_hours hours.
    """
    minutes = convert_to_minutes(num_hours)
    seconds = minutes * 60
    return seconds

seconds = convert_to_seconds(2)
```

Python defines the first two functions, creates objects for them in the heap, and, in the stack frame for the main program, creates variables that refer to those function objects.



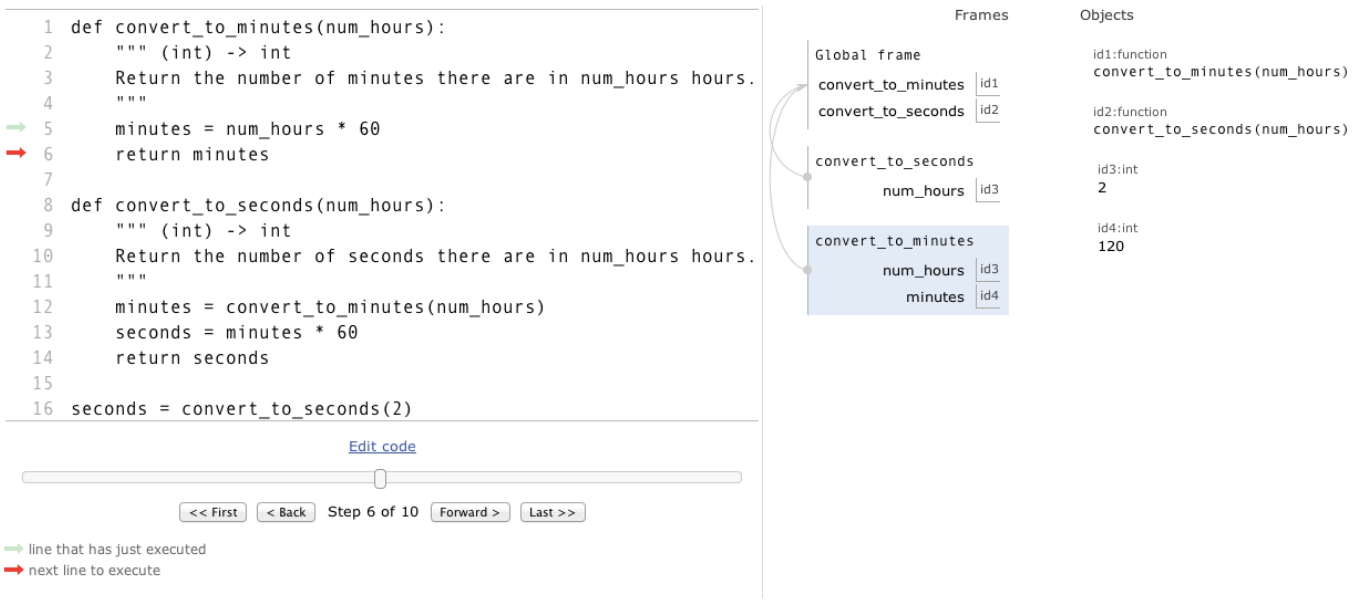
After that, it executes the assignment statement on line 16. The right-hand side of the assignment statement is a function call so we evaluate the argument, `2`, first. The frame for `convert_to_seconds` will appear on the call stack. The parameter, `num_hours`, will refer to the value `2`.



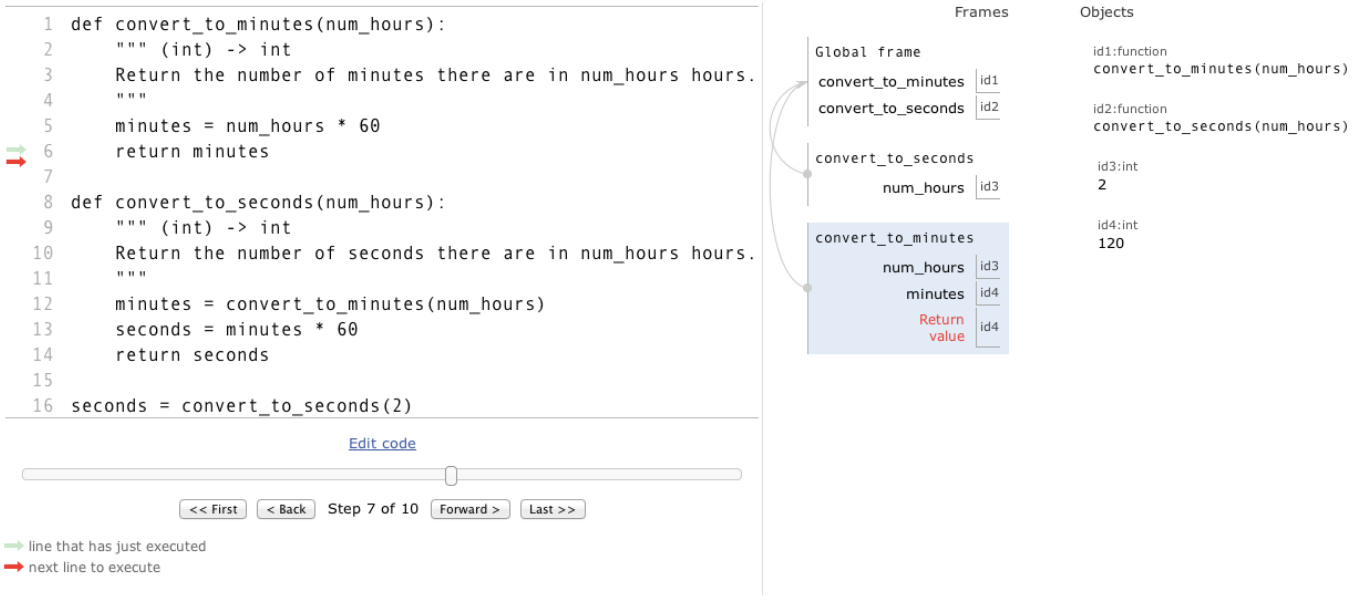
The first statement in function `convert_to_seconds` is an assignment statement. Again, we evaluate the expression on the right-hand side. This is a function call so we evaluate the argument, `num_hours`. This produces the value 2. A stack frame for function `convert_to_minutes` is created on the call stack. Python stores the memory address of 2 in the parameter for `convert_to_minutes`, which also happens to be called `num_hours`.

We now see that there are two variables called `num_hours` in the call stack; one is in `convert_to_minutes` and the other is in `convert_to_seconds`.

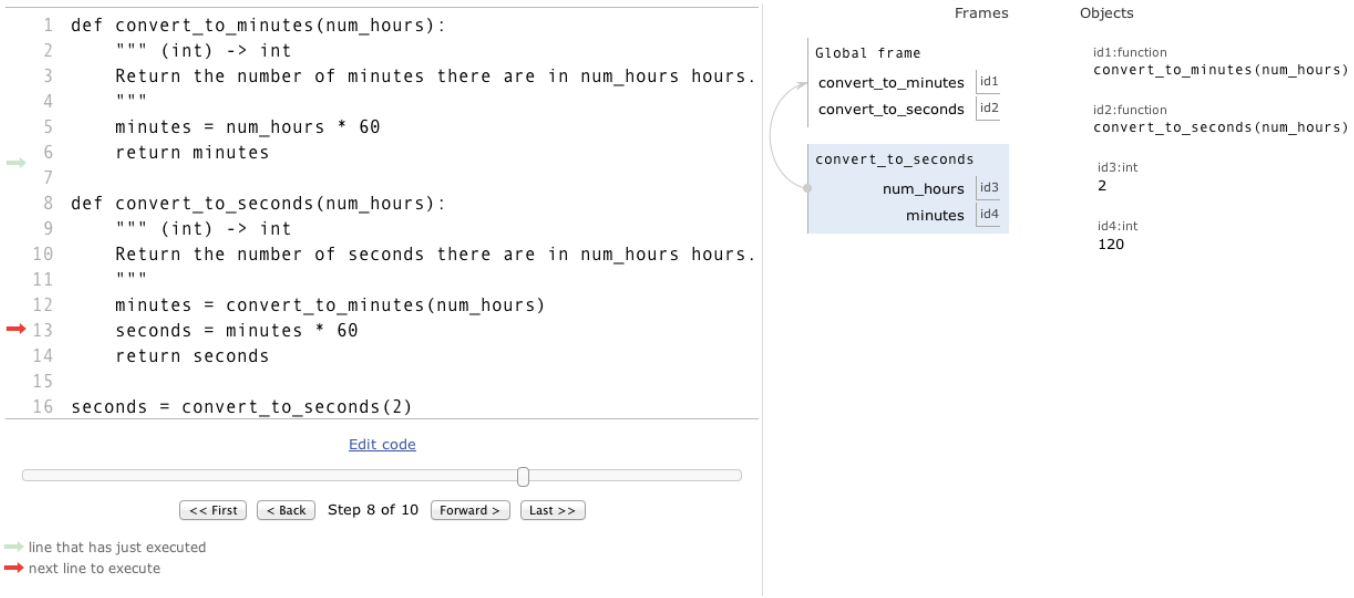
The next line of code Python executes is minutes `= num_hours * 60`. However, which instance of num_hours will be used? Python always uses the variable in the current stack frame. With an assignment statement, if the variable does not exist in the current stack frame, Python creates it. So, once `num_hours * 60` is evaluated, variable minutes is created in the current stack frame.



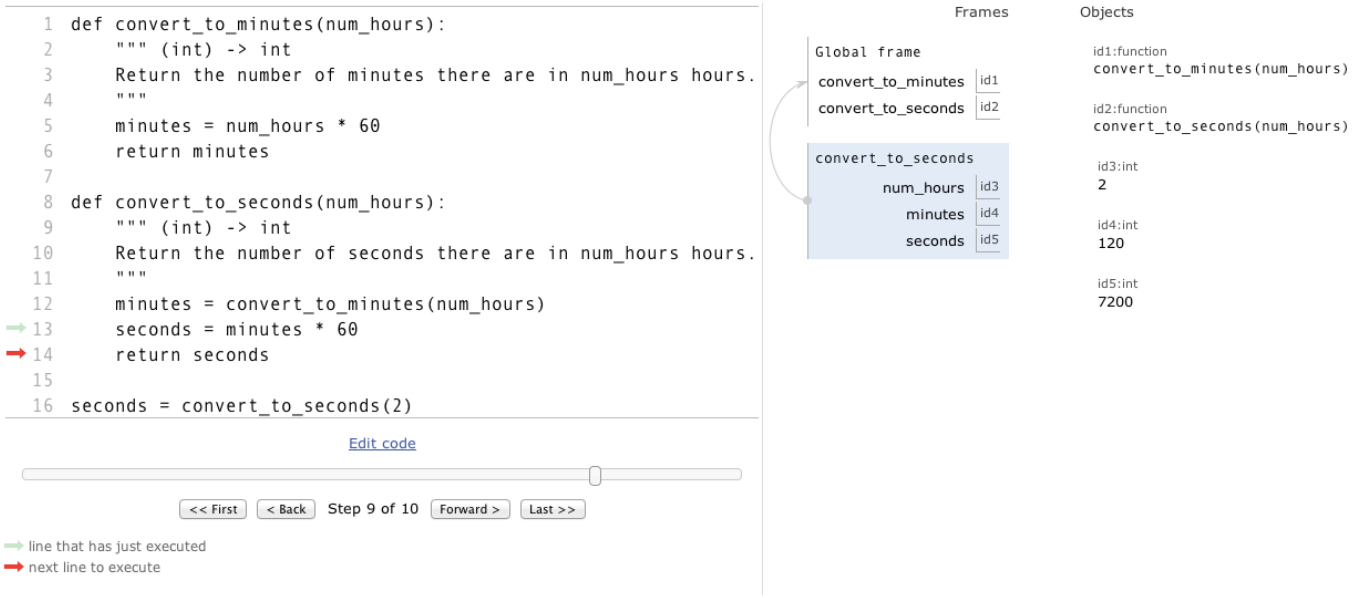
The last line of the function is return minutes. Once this statement is complete, Python will return to the frame just underneath the top of the call stack.



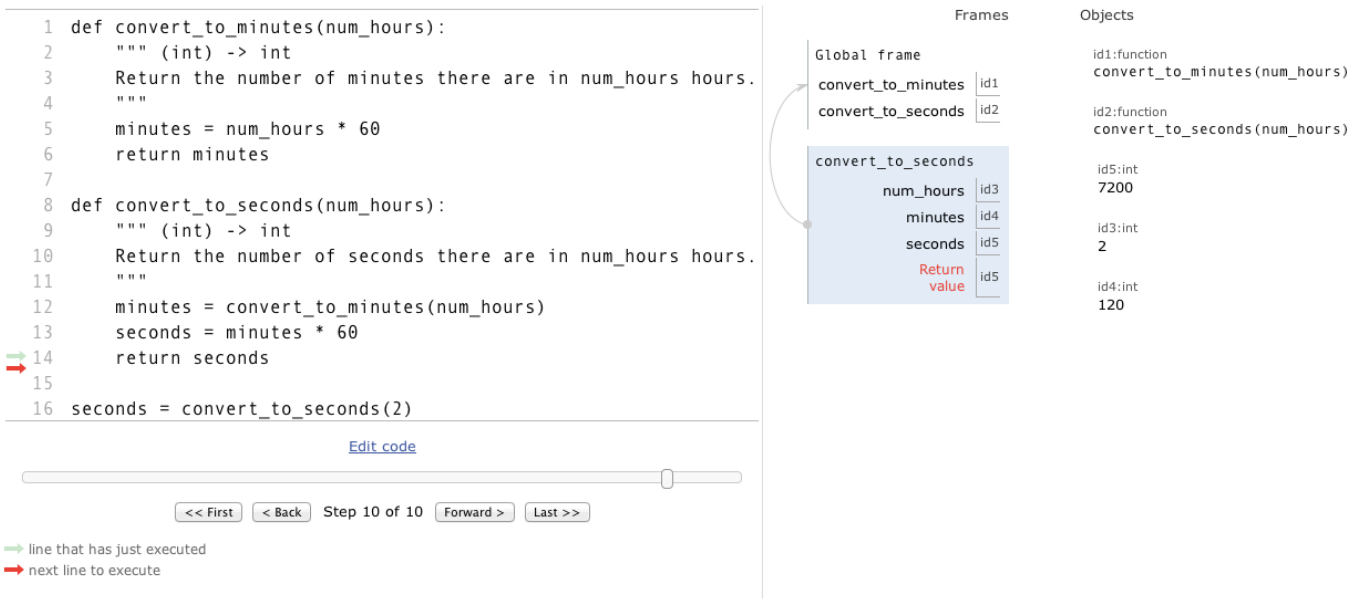
So, Python is going to produce the value 120, remove the current stack frame, create a new variable called minutes in the stack frame for `convert_to_seconds`, and store the memory adress of 120 in that variable.



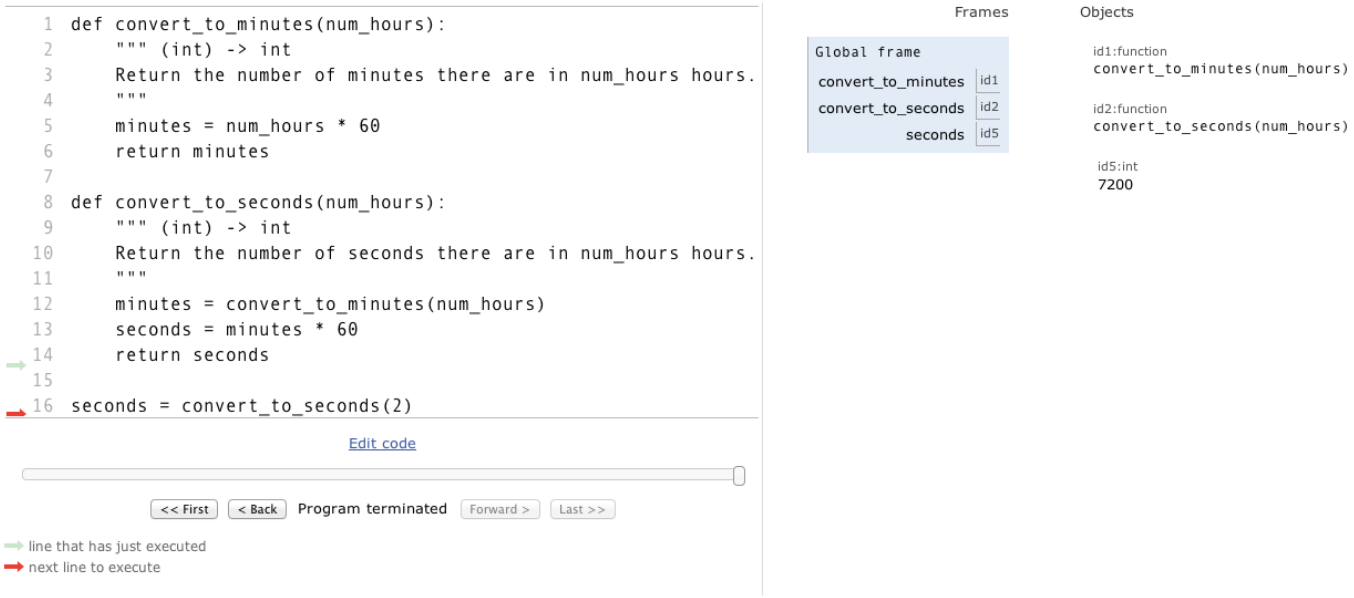
Python then executes `seconds = minutes * 60`. Python evaluates the right-hand side, which produces 7200, and stores the memory address of that value in variable seconds. Since this variable does not exist yet, Python creates it in the current stack frame.



Next is a return statement. Like we saw above, that is going to return control back to the the main module.



Once the frame for `convert_to_seconds` is removed, the assignment statement on line 16 (which has been paused a long time!) is completed, and a new variable seconds is created in the stack frame for the main program.



Notes and assignment and return statements

Assignment statement and computer memory

```
variable = expression
```

If a variable does not exist in the current stack frame, Python creates it.

Return statement and computer memory

```
return expression
```

In addition to evaluating the expression and yielding its value, `return` also erases the stack frame on top of the call stack.

Optional Book Reading

3.5 Tracing Function Calls in the Memory Model

Video 2: Type bool: Booleans in Python

Boolean values

The Python type `bool` has two values: `True` and `False`.

Comparison operators

The comparison operators take two values and produce a Boolean value.

Description	Operator	Example	Result of example
less than	<	3 < 4	True
greater than	>	3 > 4	False
equal to	==	3 == 4	False
greater than or equal to	>=	3 >= 4	False
less than or equal to	<=	3 <= 4	True
not equal to	!=	3 != 4	True

Logical operators

There are also three logical operators that produce Boolean values: `and`, `or`, and `not`.

Description	Operator	Example	Result of example
not	not	not (80 >= 50)	False
and	and	(80 >= 50) and (70 <= 50)	False
or	or	(80 >= 50) or (70 <= 50)	True

The `and` Logic Table

The and operator produces `True` if and only if both expressions are `True`.

As such, if the first operand is `False`, the second condition will not even be checked, because it is already known that the expression will produce `False`.

expr1	expr2	expr1 and expr2
True	True	True
True	False	False
False	True	False
False	False	False

The `or` Logic Table

The `or` operator evaluates to `True` if and only if at least one operand is `True`.

As such, if the first operand is `True`, the second condition will not even be checked, because it is already known that the expression will produce `True`.

expr1	expr2	expr1 or expr2

True	True	True
True	False	True
False	True	True
False	False	False

The `not` Logic Table

The not operator evaluates to True if and only if the operand is False.

expr1	**not expr1**
True	False
False	True

Double-negation can be simplified. For example, the expression not not `(4 == 5)` can be simplified to `4 == 5`.

Order of Precedence for Logical Operators

The order of precedence for logical operators is: `not`, `and`, then `or`. We can override precedence using parentheses and parentheses can also be added to make things easier to read and understand.

For example, the `not` operator is applied before the `or` operator in the following code:

```
>>> grade = 80
>>> grade2 = 90
>>> not grade >= 50 or grade2 >= 50
True
```

Parentheses can be added to make this clearer: `(not grade >= 50) or (grade2 >= 50)`

Alternatively, parentheses can be added to change the order of operations: `not ((grade >= 50) or (grade2 >= 50))`

Optional Book Reading

Chapter 5.1 A Boolean Type

Video 3: Converting between `int`, `str`, and `float`

str

Builtin function `str` takes any value and returns a string representation of that value.

```
>>> str(3)
'3'
>>> str(47.6)
'47.6'
```

int

Builtin function `int` takes a string containing only digits (possibly with a leading minus sign -) and returns the `int` that represents. Function `int` also converts `float` values to integers by throwing away the fractional part.

```
>>> int('12345')
12345
>>> int('-998')
-998
>>> int(-99.9)
-99
```

If function `int` is called with a string that contains anything other than digits, a `ValueError` happens.

```
>>> int('-99.9')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '-99.9'
```

float

Builtin function `float` takes a string containing only digits and zero or one decimal points (possibly with a leading minus sign -) and returns the `float` that represents. Function `float` also converts `int` values to `floats`.

```
>>> float('-43.2')
-43.2
>>> float('432')
432.0
>>> float(4)
4.0
```

If function `float` is called with a string that can't be converted, a ValueError happens.

```
>>> float('-9.9.9')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '-9.9.9'
```

Optional Book Reading

Chapter 4.1 Creating Strings of Characters

Video 4: Import: Using Non-Builtin Functions

Modules

Python contains many functions, but not all of them are immediately available as builtin functions. Instead of being available as builtins, some functions are saved in different modules. A *module* is a file containing function definitions and other statements.

We may also define our own modules with our own functions.

import

In order to gain access to the functions in a module, we must import that module.

The general form of an import statement is:

`import module_name`

To access a function within a module, we use:

`module_name.function_name`

For example, we can import the Python module math and call the function sqrt from it:

`import math`

```
def area2(side1, side2, side3):
    semi = semiperimeter(side1, side2, side3)
    area = math.sqrt(semi * (semi - side1) * (semi - side2) * (semi - side3))
    return area
```

In addition to importing Python's modules, we can also import the modules that we write. For example, to use the functions from `triangle.py` (from the video) in another module, we would `import triangle`. A module being imported should be in the same directory as the module importing it.

Optional Book Reading

Chapter 6.1 Importing Modules
Chapter 6.2 Defining Your Own Modules

Video 5: The `if` statement

If statements can be used to control which instructions are executed. Here is the general form:

```
if expression1:
    body1
[elif expression2:      #0 or more clauses
    body2]
[else:                  #0 or 1 clause
    bodyN]
```

`elif` stands for "else if", so this forms a chain of conditions.

To execute an `if` statement, evaluate each expression in order from top to bottom. If an expression produces `True`, execute the body of that clause and then skip the rest open 3he `if` statement. If there is an `else`, and none of the expressions produce `True`, then execute the body of the `else`.

For example, given this function:

```
def report_status(scheduled_time, estimated_time):
    """ (float, float) -> str """
    if scheduled_time == estimated_time:
        return 'on time'
    elif scheduled_time > estimated_time:
        return 'early'
    else:
        return 'delayed'
```

In the shell:

```
>>> report_status(14.3, 14.3)
'on time'
>>> report_status(12.5, 11.5)
'early'
>>> report_status(9.0, 9.5)
'delayed'
```

A note on `None`

When execution of a function body ends without having executed a `return` statement, the function returns value `None`. The type of `None` is `NoneType`.

For example, consider this function:

```
def report_status(scheduled_time, estimated_time):
    """ (float, float) -> str

    Return the flight status (on time, early, delayed) for a flight that was
    scheduled to arrive at scheduled_timed, but is now estimated to arrive
    at estimated_time.

    Pre-condition: 0.0 <= scheduled_time < 24.0 and 0.0 <= estimated_time < 24.0

    >>> report_status(14.3, 14.3)
    'on_time'
    >>> report_status(12.5, 11.5)
    'early'
    >>> report_status(9.0, 9.5)
```

```
    'delayed'
    """

    if scheduled_time == estimated_time:
        return 'on time'
```

In the shell:

```
>>> report_status(14,3, 14.3)
'on time'
>>> report_status(12.5, 11.5)
>>> print(report_status(12.5, 11.5))
None
```

Because the type of None is `NoneType`, not `str`, this breaks the Type Contract. To fix this, we would need to complete the rest of the function.

Optional Book Reading

Chapter 5.2 Choosing Which Statements to Execute Chapter 3.8 Omitting a Return Statement: None

Video 6: No `if` Required

It is common for new programmers to write code like the following:

```
def is_even(num):
    """ (int) -> bool
    Return whether num is even.
    """

    if num % 2 == 0:
        return True
    else:
        return False
```

This works, but is stylistically questionable. It's also more typing and reading than is necessary!

`num % 2 == 0` already produces `True` or `False`, so that expression can be used with the `return` statement:

```
def is_even(num):
    """ (int) -> bool
    Return whether num is even.
    """

    return num % 2 == 0
```

Optional Book Reading

Chapter 5.4 Remembering the Results of a Boolean Expression Evaluation (not a perfect match, but it didn't fit better elsewhere)

Video 7: Structuring `if` Statements

`if-elif` vs. `if-if`

An `if` statement with an `elif` clause is a single statement. The expressions are evaluated from top to bottom until one produces `True` or until there are no expressions left to evaluate. When an expression produces `True`, the body associated with it is executed and then the `if` statement exits. Any subsequent expressions are ignored. For example:

```
grade1 = 70
grade2 = 80

if grade1 >= 50:
    print('You passed a course with grade: ', grade1)
elif grade2 >= 50:
```

```
    print('You passed a course with grade: ', grade2)
```

The `if` statement condition (`grade1 >= 50`) evaluates to `True`, so the body associated with the `if` is executed and then the `if` exits. The `elif` condition is not even evaluated in this case.

It is possible for `if` statements to appear one after another in a program. Although they are be adjacent to each other, they are completely independent of each other and it is possible for the body of each if to be executed. For example:

```
grade1 = 70
grade2 = 80

if grade1 >= 50:
    print('You passed a course with grade: ', grade1)
if grade2 >= 50:
    print('You passed a course with grade: ', grade2)
```

In the program above, the condition associated with the first `if` statement (`grade1 >= 50`) produces `True`, so the body associated with it is executed. The condition associated with the second `if` statement (`grade2 >= 50`) also produces `True`, so the body associated with it is also executed.

Nested `ifs`

It is possible to place an `if` statement within the body of another `if` statement. For example:

```
if precipitation:
    if temperature > 0:
        print('Bring your umbrella!')
    else:
        print('Wear your snow boots and winter coat!)
```

The statement above can be simplified by removing some of the nesting. The message `'Bring your umbrella!'` is printed only when both of the `if` statement conditions are `True`. The message `'Wear your snow boots and winter coat!'` is printed only when the outer `if` condition is `True`, but the inner `if` condition is `False`. The following is equivalent to the code above:

```
if precipitation and temperature > 0:
    print('Bring your umbrella')
elif precipitation:
    print('Wear your snow boots and winter coat!')
```

Optional Book Reading

Chapter 5.3 Nested If Statements