While, While West

Floor division and modulus

The **floor division** operator, `//`, divides two numbers and rounds down to an integer.

```
>>> minutes = 105
>>> minutes / 60
1.75
>>> minutes // 60
1
```

The **modulus operator**, `%`, divides two numbers and returns the remainder.

```
>>> minutes % 60
45
```

Modulus

The modulus operator is more useful than you might think.

You can check if one number is divisible by another: if `x % y` is 0, then you know that `x` is divisible by `y`.

```
def is_even(number):
  return number % 2 == 0
```

Keyboard input

Python provides a built-in function called `input` that stops the program and waits for the user to type something. When the user presses **Return** or **Enter**, the program resumes and `input` returns what the user typed as a string.

`input` is a fruitful function—it returns what the user typed. You will almost always want to store its return value in a variable.

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

You usually want to tell the user what to type. You can pass in a prompt as the argument to `input`.

```
>>> first_number = input('Enter a number: ')
Enter a number: 3
```

Keyboard input

You can use the newline special character, `\n`, if you would like the user to enter input on a new line.

```
>>> first_number = input('Enter a number:\n')
Enter a number:
3
```

Remember that `input` returns the user input as a string. If you want to use the user input for a numerical computation, you need to first convert the return value into the appropriate data type.

```
>>> first_number = input('Enter a number: ')
Enter a number: 3
>>> second_number = input('Enter another number: ')
Enter another number: 2
>>> first_number + second_number
'32'
```

Boolean expressions

A **boolean expression** is an expression that is either true or false.

```
>>> 5 == 5
True
>>> 5 == 6
False
```

These boolean expressions use the `==` **relational operator**, which produces `True` if the operands are equal and `False` if they are not.

Boolean expressions

There are other relational operators that can be used in boolean expressions.

Expression	Description
`x != y`	x is not equal to y
`x > y`	x is greater than y
`x < y`	x is less than y
`x >= y`	x is greater than or equal to y
`x <= y`	x is less than or equal to y

Relational operators: A couple things to note...

A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a relational operator.

```
>>> x = 2
>>> y = 3
>>> x = y
>>> x
3
```

There is no such thing as `=<` or `=>`.

```
>>> x = 2
>>> y = 3
>>> x =< y
File "<stdin>", line 1
  x =< y
     ^
SyntaxError: invalid syntax
```

Try to remember the phrases "less than or equal to" `<=` and "greater than or equal to" `>=`.

Logical operators

Logical operators are used to *combine* boolean expressions.

In Python, the logical operators are `and`, `or`, and `not`.

The meaning of these logical operators in programming are similar to their meaning in English.

`and`: `x > 0 and x < 10` is true only if `x` is greater than 0 *and* less than 10.

`or`: `n % 2 == 0 or n % 3 == 0` is true if *either or both* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

`not`: `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

Logical operators

`7 > 2 and 7 % 2 == 0` `False`

`8 - 2 < 4 or True` `True`

`not False` `True`

`True or 2 + 3 > x` `True`

`3 % 4 == 0 or (7 % 2 == 1 and 12 % 2 == 1)` `False`

Boolean type

Remember that all boolean expressions evaluate to `True` or `False`. `True` and `False` are special values that belong to the type bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Conditional execution

In almost all useful programs, we need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability.

The simplest conditional statement is the `if` statement:

```
if x > 0:
  print('x is positive')
```

The boolean expression after `if`—`x > 0`—is the **condition.** If it is true, the indented statement runs. If it is false, nothing happens. There is no limit on how many indented statements you can have, but you must have at least one.

Conditional execution

Check if `x` is an even number. If it is, print its value and that it is even.

```
x = 2
```

```
def is_even(number):
  return number % 2 == 0

x = 2
```

Check if `y` is an an odd number. If it is, print its value and that it is odd.

```
def is_even(number):
  return number % 2 == 0

y = 3
```

Alternative execution

A second form of the `if` statement is **alternative execution**, in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

Nested conditionals

One conditional can also be nested within another. We could have written the example in the previous slide like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

```
if 0 < x and x < 10: # or even `if 0 < x < 10` in this case
    print('x is a positive single-digit number.')
```

Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

If `n` is 0 or negative, it outputs the word, "Blastoff!". Otherwise, it prints `n` and then calls `countdown` (itself). The condition(s) where the function no longer calls itself is referred to as the **base case**. So the base case of `countdown` is when `n` is 0 or negative. Here is the output of calling `countdown(3)`:

```
3
2
1
Blastoff!
```

The `while` statement

Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called iteration. We have already seen one function, `countdown`, that iterates using recursion. Another way of iterating is the `while` statement. Here is a version of `countdown` that uses a `while` statement:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

You can almost read this `while` statement as if it were English. It means, "While `n` is greater than 0, display the value of `n` and then decrement `n`. When you get to 0, display the word `Blastoff!`"

The `while` statement

More formally, here is the flow of execution for a `while` statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of execution flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

The `break` statement

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the `break` statement to jump out of the loop.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

```
> not done
not done
> done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").