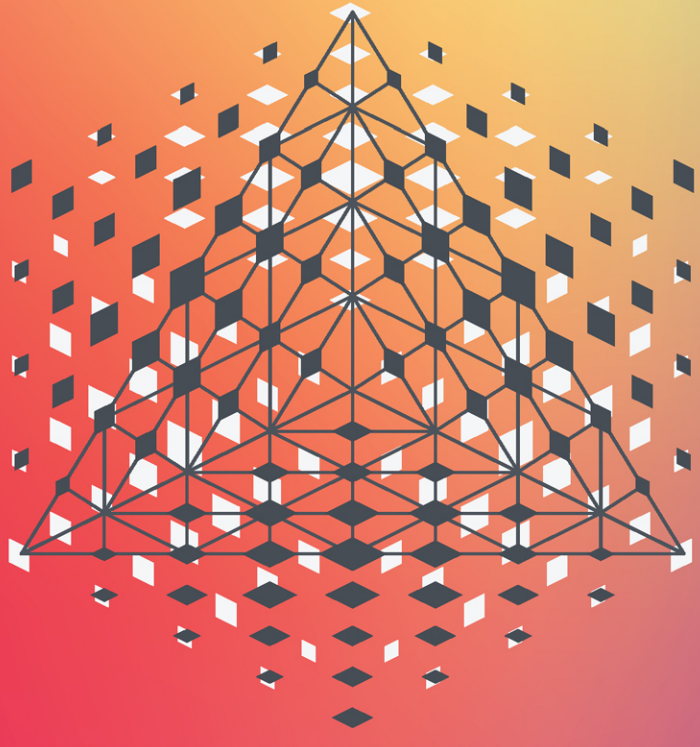
ADDISON WESLEY DATA & ANALYTICS SERIES

FOUNDATIONAL PYTHON FOR DATA SCIENCE



KENNEDY BEHRMAN

Foundational Python for Data Science

Listing 5.1 **Equality Operations**

```
# Assign values to variables
a, b, c = 1, 1, 2
# Check if value is equal
a == b
True

a == c
False

a != b
False

a != c
True
```

You can compare different types of objects by using the equality/inequality operators. For numeric types, such as floats and integers, the values are compared. For example, if you compare the integer `1` to the float `1.0`, they evaluate as equal:

```
1 == 1.0
True
```

Most other cross-type comparisons return `False`, regardless of value. Comparing a string to an integer will always return `False`, regardless of the values:

```
'1' == 1
False
```

Web forms often report all user input as strings. A common problem occurs when trying to compare user input from a web form that represents a number but is of type string with an actual number. String input always evaluates to `False` when compared to a number, even if the input is a string version of the same value.

Comparison Operations

You use comparison operators to compare the order of objects. What "the order" means depends on the type of objects compared. For numbers, the comparison is the order on a number line, and for strings, the Unicode value of the characters is used. The comparison operators are less than (`<`), less than or equal to (`<=`), greater than (`>`), and greater than or equal to (`>=`). Listing 5.2 demonstrates the behavior of various comparison operators.

Listing 5.2 **Comparison Operations**

```
a, b, c = 1, 1, 2
a < b
False
```

```
a < c
True

a <= b
True

a > b
False

a >= b
True
```

There are certain cases where you can use comparison operators between objects of different types, such as with the numeric types, but most cross-type comparisons are not allowed. If you use a comparison operator with noncomparable types, such as a string and a list, an error occurs.

Boolean Operations

The Boolean operators are based on Boolean math, which you may have studied in a math or philosophy course. These operations were first formalized by the mathematician George Boole in the 19th century. In Python, the Boolean operators are `and`, `or`, and `not`. The `and` and `or` operators each take two arguments; the `not` operator takes only one.

The `and` operator evaluates to `True` if both of its arguments evaluate to `True`; otherwise, it evaluates to `False`. The `or` operator evaluates to `True` if either of its arguments evaluates to `True`; otherwise, it evaluates to `False`. The `not` operator returns `True` if its argument evaluates to `False`; otherwise, it evaluates to `False`. Listing 5.3 demonstrates these behaviors.

Listing 5.3 **Boolean Operations**

```
True and True
True

True and False
False

True or False
True

False or False
False

not False
True

not True
False
```

Both the and and or operators are short-circuit operators. This means they will only evaluate their input expression as much as is needed to determine the output. For example, say that you have two methods, returns_false() and returns_true(), and you use them as inputs to the and operator as follows:

```
returns_false() and returns_true()
```

If returns_false() returns False, returns_true() will not be called, as the result of the and operation is already determined. Similarly, say that you use them as arguments to the or operation, like this:

```
returns_true() or returns_false()
```

In this case, the second method will not be called if the first returns True.

The not operator always returns one of the Boolean constants True or False. The other two Boolean operators return the result of the last expression evaluated. This is very useful with object evaluation.

Object Evaluation

All objects in Python evaluate to True or False. This means you can use objects as arguments to Boolean operations. The objects that evaluate to False are the constants None and False, any numeric with a value of zero, or anything with a length of zero. This includes empty sequences, such as an empty string ("") or an empty list ([]). Almost anything else evaluates to True.

Because the or operator returns the last expression it evaluates, you can use it to create a default value when a variable evaluates to False:

```
a = ''
b = a or 'default value'
b
'default value'
```

Because this example assigns the first variable to an empty string, which has a length of zero, this variable evaluates to False. The or operator evaluates this and then evaluates and returns the second expression.

if Statements

The if statement is a compound statement. if statements let you branch the behavior of your code depending on the current state. You can use an if statement to take an action only when a chosen condition is met or use a more complex one to choose among multiple actions, depending on multiple conditions. The control statement starts with the keyword if followed by an expression (which evaluates to True or False) and then a colon. The controlled statements follow either on the same line separated by semicolons:

```
if True:message="It's True!";print(message)
It's True!
```

or as an indented block of code, separated by newlines:

```
if True:
    message="It's True"
    print(message)
It's True
```

In both of these examples, the controlling expression is simply the reserved constant `True`, which always evaluates to `True`. There are two controlled statements: The first assigns a string to the variable message, and the second prints the value of this variable. It's usually more readable to use the block syntax, as in the second example.

If the controlling expression evaluates to `False`, the program continues executing and skips the controlled statement(s):

```
if False:
    message="It's True"
    print(message)
```

The Walrus Operator

When you assign a value to a variable, Python does not return a value. A common situation is to make a variable assignment and then check the value of the variable. For example, you might assign to a variable the value returned by a function, and if that value is not `None`, you may use the returned object. The search method of the Python `re` module (covered in Chapter 15, "Other Topics") returns a `match` object if it finds a match in a string, and it returns `None` otherwise, so if you want to use the `match` object, you need to make sure it's not `None` first:

```
import re
s = '2020-12-14'
match = re.search(r'(\d\d\d\d)-(\d\d)-(\d\d)', s)
if match:
    print(f"Matched items: {match.groups(1)}")
else:
    print(f"No match found in {s}")
```

Python 3.8 introduced a new operator, the assignment operator (`:=`). It is referred to as the *walrus operator* due to its resemblance to a walrus's head. This operator assigns a value to a variable and returns that value. You could rewrite the `match` example by using it:

```
import re
s = '2020-12-14'
if match := re.search(r'(\d\d\d\d)-(\d\d)-(\d\d)', s):
    print(f"Matched items: {match.groups(1)}")
else:
    print(f"No match found in {s}")
```

This operator creates less complicated, more readable code.

Here is an example that uses a membership test as the controlling expression:

```
snack = 'apple'
fruit = {'orange', 'apple', 'pear'}
if snack in fruit:
   print(f"Yeah, {snack} is good!"
Yeah, apple is good!
```

This example checks whether the value of the variable `snack` is in the set `fruit`. If it is, an encouraging message is printed.

If you want to run an alternative block of code when the controlling expression is `False`, you can use an `else` statement. An `else` statement consists of the keyword `else` followed by a colon and then a block of code that will execute only if the controlling expression preceding it evaluates to `False`. This lets you branch the logic in your code. Think of it as choosing which actions to take based on the current state. Listing 5.4 shows an `else` statement added to the snack-related `if` statement. The second print statement executes only if the controlling expression `snack` in `fruit` is `False`.

Listing 5.4 **`else` Statements**

```
snack = 'cake'
fruit = {'orange', 'apple', 'pear'}
if snack in fruit:
   print(f"Yeah, {snack} is good!")
else:
   print(f"{snack}!? You should have some fruit")
cake!? You should have some fruit
```

If you want to have multiple branches in your code, you can nest `if` and `else` statements as shown in Listing 5.5. In this case, three choices are made: one if the balance is positive, one if it is negative, and one if it is negative.

Listing 5.5 **Nested `else` Statements**

```
balance = 2000.32
account_status = None

if balance > 0:
    account_status = 'Positive'
else:
   if balance == 0:
       account_status = 'Empty'
   else:
       account_status = 'Overdrawn'

print(account_status)
Positive
```

While this code is legitimate and will work the way it is supposed to, it is a little hard to read. To perform the same branching logic in a more concise way, you can use an `elif` statement. This type of statement is added after an initial `if` statement. It has a controlling expression of its own, which will be evaluated only if the previous statement's expression evaluates to `False`. Listing 5.6 performs the same logic as Listing 5.5, but has the nested `else` and `if` statements replaced by `elif`.

Listing 5.6 **`elif` Statements**

```
balance = 2000.32
account_status = None

if balance > 0:
    account_status = 'Positive'
elif balance == 0:
    account_status = 'Empty'
else:
    account_status = 'Overdrawn'

print(account_status)
Positive
```

By chaining multiple `elif` statements with an `if` statement, as demonstrated in Listing 5.7, you can perform complicated choices. Usually an `else` statement is added at the end to catch the case that all the controlling expressions are `False`.

Listing 5.7 **Chaining `elif` Statements**

```
fav_num = 13

if fav_num in (3,7):
    print(f"{fav_num} is lucky")
elif fav_num == 0:
    print(f"{fav_num} is evocative")
elif fav_num > 20:
    print(f"{fav_num} is large")
elif fav_num == 13:
    print(f"{fav_num} is my favorite number too")
else:
    print(f"I have no opinion about {fav_num}")
is my favorite number too
```

`while` Loops

A `while` loop consists of the keyword `while` followed by a controlling expression, a colon, and then a controlled code block. The controlled statement in a `while` loop executes only if the controlling statement evaluates to `True`; in this way, it is like an `if` statement. Unlike an `if`