Basic Python Semantics: Operators

In the previous section, we began to look at the semantics of Python variables and objects; here we'll dig into the semantics of the various *operators* included in the language. By the end of this section, you'll have the basic tools to begin comparing and operating on data in Python.

Arithmetic Operations

Python implements seven basic binary arithmetic operators, two of which can double as unary operators. They are summarized in the following table:

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`
`+a`	Unary plus	`a` unchanged (rarely used)

These operators can be used and combined in intuitive ways, using standard parentheses to group operations. For example:

```
# addition, subtraction, multiplication
(4 + 8) * (6.5 - 3)
```

```
42.0
```

Floor division is true division with fractional parts truncated:

```
# True division
print(11 / 2)
```

```
5.5
```

```
# Floor division
print(11 // 2)
```

```
5
```

The floor division operator was added in Python 3; you should be aware if working in Python 2 that the standard division operator (`/`) acts like floor division for integers and like true division for floating-point numbers.

Finally, I'll mention an eighth arithmetic operator that was added in Python 3.5: the `a @ b` operator, which is meant to indicate the *matrix product* of `a` and `b`, for use in various linear algebra packages.

Bitwise Operations

In addition to the standard numerical operations, Python includes operators to perform bitwise logical operations on integers. These are much less commonly used than the standard arithmetic operations, but it's useful to know that they exist. The six bitwise operators are summarized in the following table:

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both `a` and `b`
a | b	Bitwise OR	Bits defined in `a` or `b` or both
`a ^ b`	Bitwise XOR	Bits defined in `a` or `b` but not both
`a << b`	Bit shift left	Shift bits of `a` left by `b` units
`a >> b`	Bit shift right	Shift bits of `a` right by `b` units
`~a`	Bitwise NOT	Bitwise negation of `a`

These bitwise operators only make sense in terms of the binary representation of numbers, which you can see using the built-in `bin` function:

```
bin(10)
```

```
'0b1010'
```

The result is prefixed with `'0b'`, which indicates a binary representation. The rest of the digits indicate that the number 10 is expressed as the sum $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Similarly, we can write:

```
bin(4)
```

```
'0b100'
```

Now, using bitwise OR, we can find the number which combines the bits of 4 and 10:

```
4 | 10
```

```
14
```

```
bin(4 | 10)
```

```
'0b1110'
```

These bitwise operators are not as immediately useful as the standard arithmetic operators, but it's helpful to see them at least once to understand what class of operation they perform. In particular, users from other languages are sometimes

tempted to use XOR (i.e., `a ^ b`) when they really mean exponentiation (i.e., `a ** b`).

Assignment Operations

We've seen that variables can be assigned with the "`=`" operator, and the values stored for later use. For example:

```
a = 24
print(a)
```

```
24
```

We can use these variables in expressions with any of the operators mentioned earlier. For example, to add 2 to `a` we write:

```
a + 2
```

```
26
```

We might want to update the variable `a` with this new value; in this case, we could combine the addition and the assignment and write `a = a + 2`. Because this type of combined operation and assignment is so common, Python includes built-in update operators for all of the arithmetic operations:

```
a += 2  # equivalent to a = a + 2
print(a)
```

```
26
```

There is an augmented assignment operator corresponding to each of the binary operators listed earlier; in brief, they are:

`a += b`	`a -= b`
`a //= b`	`a %= b`
a |= b	`a ^= b`

Each one is equivalent to the corresponding operation followed by assignment: that is, for any operator "`?`", the expression `a ?= b` is equivalent to `a = a ? b`, with a slight catch. For mutable objects like lists, arrays, or DataFrames, these augmented assignment operations are actually subtly different than their more verbose counterparts: they modify the contents of the original object rather than creating a new object to store the result.

Comparison Operations

Another type of operation which can be very useful is comparison of different values. For this, Python implements standard comparison operators, which

return Boolean values `True` and `False`. The comparison operations are listed in the following table:

| Operation | Description || Operation | Description | |---|---||---|---| | `a == b` | `a` equal to `b` || `a != b` | `a` not equal to `b` | | `a < b` | `a` less than `b` || `a > b` | `a` greater than `b` | | `a <= b` | `a` less than or equal to `b` || `a >= b` | `a` greater than or equal to `b` |

These comparison operators can be combined with the arithmetic and bitwise operators to express a virtually limitless range of tests for the numbers. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

```
# 25 is odd
25 % 2 == 1
```

```
True
```

```
# 66 is odd
66 % 2 == 1
```

```
False
```

We can string-together multiple comparisons to check more complicated relationships:

```
# check if a is between 15 and 30
a = 25
15 < a < 30
```

```
True
```

And, just to make your head hurt a bit, take a look at this comparison:

```
-1 == ~0
```

```
True
```

Recall that `~` is the bit-flip operator, and evidently when you flip all the bits of zero you end up with -1. If you're curious as to why this is, look up the *two's complement* integer encoding scheme, which is what Python uses to encode signed integers, and think about happens when you start flipping all the bits of integers encoded this way.

Boolean Operations

When working with Boolean values, Python provides operators to combine the values using the standard concepts of "and", "or", and "not". Predictably, these operators are expressed using the words `and`, `or`, and `not`:

```
x = 4
(x < 6) and (x > 2)
```

```
True
```

```
(x > 10) or (x % 2 == 0)
```

```
True
```

```
not (x < 6)
```

```
False
```

Boolean algebra aficionados might notice that the XOR operator is not included; this can of course be constructed in several ways from a compound statement of the other operators. Otherwise, a clever trick you can use for XOR of Boolean values is the following:

```
# (x > 1) xor (x < 10)
(x > 1) != (x < 10)
```

```
False
```

These sorts of Boolean operations will become extremely useful when we begin discussing *control flow statements* such as conditionals and loops.

One sometimes confusing thing about the language is when to use Boolean operators (`and`, `or`, `not`), and when to use bitwise operations (`&`, `|`, `~`). The answer lies in their names: Boolean operators should be used when you want to compute *Boolean values (i.e., truth or falsehood) of entire statements*. Bitwise operations should be used when you want to *operate on individual bits or components of the objects in question*.

Identity and Membership Operators

Like `and`, `or`, and `not`, Python also contains prose-like operators to check for identity and membership. They are the following:

Operator	Description
`a is b`	True if `a` and `b` are identical objects
`a is not b`	True if `a` and `b` are not identical objects
`a in b`	True if `a` is a member of `b`
`a not in b`	True if `a` is not a member of `b`

Identity Operators: "`is`" and "`is not`"

The identity operators, "`is`" and "`is not`" check for *object identity*. Object identity is different than equality, as we can see here:

```
a = [1, 2, 3]
b = [1, 2, 3]
```

```
a == b
```

```
True
```

```
a is b
```

```
False
```

```
a is not b
```

```
True
```

What do identical objects look like? Here is an example:

```
a = [1, 2, 3]
b = a
a is b
```

```
True
```

The difference between the two cases here is that in the first, `a` and `b` point to *different objects*, while in the second they point to the *same object*. As we saw in the previous section, Python variables are pointers. The "`is`" operator checks whether the two variables are pointing to the same container (object), rather than referring to what the container contains. With this in mind, in most cases that a beginner is tempted to use "`is`" what they really mean is `==`.

Membership operators

Membership operators check for membership within compound objects. So, for example, we can write:

```
1 in [1, 2, 3]
```

```
True
```

```
2 not in [1, 2, 3]
```

```
False
```

These membership operations are an example of what makes Python so easy to use compared to lower-level languages such as C. In C, membership would generally be determined by manually constructing a loop over the list and checking for equality of each value. In Python, you just type what you want to know, in a manner reminiscent of straightforward English prose.