Expressions

Languages and Computation

Every powerful language has three mechanisms for combining simple ideas to form more complex ideas:(SICP 1.1)

- *primitive expressions*, which represent the simplest entities the language is concerned with,
- *means of combination*, by which compound elements are built from simpler ones, and
- *means of abstraction*, by which compound elements can be named and manipulated as units.

In the last lesson we learned about values and variables, and introduced compound expressions. In this lesson we'll dive more deeply into compound expressions.

Statements vs Expressions

Expressions have values, statements only have effects – like binding a variable to a value or effecting control flow in a program. Assignment using `=` is a statement – it cannot be used where a value is expected.

```
>>> while guess = input("Guess a number: "):
...     if guess == "7":
...         break
  Input In [42]
    while guess = input("Guess a number: "):
          ^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Python 3.8 introduced the "walrus" operator, `:=`, which creates an assignment expression, where the assigned value is the value of the expression:

```
>>> while guess := input("Guess a number: "):
...     if guess == "7":
...         break
...
```

Note that `while` and `if` are statements – they don't produce values, they create effects.

We will see a few cases in future lessons where the walrus operator is helpful.

Expression Evaluation

Mathematical expressions are evaluated using precedence and associativity rules as you would expect from math:

```
>>> 2 + 4 * 10
42
```

If you want a different order of operations, use parentheses:

```
>>> (2 + 4) * 10
60
```

Precedence and associativity rules apply to overloaded versions of operators as well:

```
>>> "Honey" + "Boo" * 2
'HoneyBooBoo'
```

Active Review

- How could we modify the expression above to evaluate to 'HoneyBooHoneyBoo' ?

Python is Strongly Typed

Python is strongly typed, meaning that Python does not allow expressions with incompatible types.

Active Review

Evaluate the following expressions in the Python REPL. Be sure to type them exactly as written.

- `2 + 3`
- `"2" + "3"`
- `"2" + 3`
- `2 + "3"`

Type Conversions

Convert a value to a different type by applying conversions named after the target type.

```
>>> int(2.9)
2
>>> float(True)
1.0
>>> int(False)
0
>>> str(True)
'True'
>>> int("False")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'False'
```

Active Review

Modify the following expressions to produce the indicated results.

- `"2" + 3` (we want `"23"`)
- `2 + "3"` (we want `5`)

Boolean Values

There are `10` kinds of people:

- those who know binary, and
- those who don't.

Python Booleans

In Python, boolean values have the `bool` type. Four kinds of boolean expressions:

- `bool` literals: `True` and `False`
- `bool` variables
- expressions formed by combining non-`bool` expressions with comparison operators
- expressions formed by combining `bool` expressions with logical operators

Comparison Operators

- Equal to: `==`, like $=$ in math
  - Remember, `=` is assignment operator, `==` is comparison operator!
- Not equal to: `!=`, like $\neq$ in math
- Greater than: `>`, like $>$ in math
- Greater than or equal to: `>=`, like $\geq$ in math

```
1 == 1 # True
1 != 1 # False
1 >= 1 # True
1 > 1  # False
```

Active Review

- What is the value of `"foo" == "Foo"`?
- What is the value of `"foo" > "Foo"`?

Logical Operators

The values produced by logical operators are often shown in truth tables:

a	b	not a	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Some examples:

```
1 True and True  # True
2 True and False # False
3 True or False  # True
4 False or False # False
5 not True       # False
```

Truth in Python

The zero values of built-in types are equivalent to `False`:

- boolean `False`
- `None`
- integer `0`
- float `0.0`
- empty string `""`
- empty list `[]`
- empty tuple `()`
- empty dict `{}`
- empty set `set()`

All other values are equivalent to True.

- Every value in Python is either *truthy* or *falsey* and can be used in a boolean context.

Short-circuit Evaluation

Logical expressions use short-circuit evaluation:

- `or` only evaluates second operand if first operand is `False`
- `and` only evaluates second operand if first operand is `True`

Guard idiom: `(b == 0)or print(a / b)`, or `(b != 0)and print(a / b)`

Active Review

What are the values of the following expressions?

- `True and False`
- `True and 0`
- `True and []`
- `True and None`
- `type(True and None)`
- `False or 1`
- `True or 1`
- `1 and "done"`
- `1 == 1 or 0`
- `1 == 1 and 0`
- `1 == (1 and 0)`

Expressions

- Expressions produce values.
- We combine values using operators and functions to form compound expressions.
- Anywhere a value is expected, a compound expression or function call can be used.