In many ways, the most important data type in Python is Boolean data, even though it is by far the simplest. The are only two values in the Boolean domain, True and False, but these are exactly the values you need if you want your program to make decisions.

The idea of Boolean data comes from the English mathematician George Boole who in 1854 wrote a book entitled *An Investigation into the Laws of Thought*. That book introduced a system of logic that has come to be known as Boolean algebra, which is the foundation for Boolean data.

### Statement Types in Python

- **Simple statements** are typically assignments or function calls.
- **Control statements** fall into two categories:
  - **Conditional statements** that specify some kind of test
  - **Iterative statements** that specify repetition
- Control statements begin with a **header line** that defines how the control statement affects the program flow.
- Each control statement takes effect over a block of statements indented, typically by four spaces, with respect to the header line. This block of statements is called the **body**.

### Boolean Expressions

- Python defines two types of operators that work with Boolean data: **relational operators** and **logical operators**.
- There are six relational operators that compare values of other types and produce a True/False result:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equals</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

For example, the expression `n <= 10` has the value True if `n` is less than or equal to 10 and the value False otherwise.

- There are also three logical operators:
  - `and` Logical AND: `p and q` means both `p` and `q`
  - `or` Logical OR: `p or q` means either `p` or `q` (or both)
  - `not` Logical NOT: `not p` means the opposite of `p`

### Notes on the Boolean Operators

- Remember that Python uses `=` for assignment. To test whether two values are equal, you must use the `==` operator.
- The `or` operator means *either or both*, which is not always clear in the English interpretation of or.
- Be careful when you combine the `not` operator with `and` and `or` because the interpretation differs from informal English.
Short-Circuit Evaluation

- Python evaluates the \texttt{and} and \texttt{or} operators using a strategy called \textit{short-circuit mode} in which it evaluates the right operand only if it needs to do so.
- For example, if \( n \) is 0, the right operand of \texttt{and} in
  \[
  \texttt{n != 0 and x \% n == 0}
  \]
  is not evaluated at all because \( n != 0 \) is \texttt{False}. Because the expression
  \[
  \texttt{False and anything}
  \]
  is always \texttt{False}, the rest of the expression no longer matters.
- One of the advantages of short-circuit evaluation is that you can use \texttt{and} and \texttt{or} to prevent errors. If \( n \) were 0 in the earlier example, evaluating \( x \% n \) would result in a division by zero.

The \texttt{if} Statement

- The simplest of the control statements is the \texttt{if} statement, which occurs in two forms. You use the first when you need to perform an operation only if a particular condition is true:
  \[
  \text{if condition: statements to be executed if the condition is true}
  \]
- You use the second form whenever you need to choose between two alternative paths, depending on whether the condition is true or false:
  \[
  \text{if condition: statements to be executed if the condition is true}
  \text{else: statements to be executed if the condition is false}
  \]

Functions Involving Control Statements

- The body of a function can contain statements of any type, including control statements. As an example, the following function uses an \texttt{if} statement to find the larger of two values:
  \[
  \text{def max(x, y): if x > y: return x else: return y}
  \]
- As this example makes clear, \texttt{return} statements can be used at any point in the function and may appear more than once.

The \texttt{while} Statement

- The \texttt{while} statement is the simplest of Python’s iterative control statements and has the following form:
  \[
  \text{while condition: statements to be repeated}
  \]
- When Python encounters a \texttt{while} statement, it begins by evaluating the condition.
- If the value of \texttt{condition} is true, Python executes the statements in the body of the loop.
- At the end of each cycle, Python reevaluates \texttt{condition} to see whether its value has changed. If \texttt{condition} is false, Python exits from the loop and continues with the statement after the end of the \texttt{while} body.

Summing the Digits in a Number

- Many operations in programming are best learned as \textit{idioms} or \textit{patterns}. A particularly useful pattern reads lines from the user until the user enters a \textit{sentinel}, typically a blank line, indicating the end of the input. The Repeat-Until-Sentinel pattern looks like this:
  \[
  \text{while True: line = input("prompt") if line == ": break rest of loop body}
  \]
- The \texttt{while True} control line loops forever. The loop exits when the \texttt{break} statement is executed at the end of the input.
- There are other ways to write this code, which is an example of the \textit{loop-and-a-half problem}, but I prefer this strategy.
The AddList Program

```python
# File: AddList.py

def AddList():
    print("This program adds a list of integers.")
    sum = 0
    while True:
        line = input(" ? ")
        if line == "":
            break
        sum += int(line)
    print("The sum is", sum)

# Startup code
if __name__ == "__main__":
    AddList()
```

The for Statement

- The for statement is Python’s most powerful mechanism for specifying iteration and has the following general form:

```
for var in iterable:
    statements to be repeated
```

- In this pattern, var is a variable name and iterable is any expression that produces a value that supports iteration.
- The effect of the for statement is to iterate over each value produced by the iterable object and assign it to var. The for loop continues as long as there are more values in the iterator.

The range Function

- The most common for loop pattern uses the built-in `range` function to produce the iterable object.
- The `range` function can take one, two, or three arguments, as follows:
  ```
  range(limit) counts from 0 up to limit – 1
  range(start, limit) counts from start up to limit – 1
  range(start, limit, step) counts by step
  ```
- Note that the `range` function always stops one step before the limit value is reached. In mathematics, a range that includes its first endpoint but not its last is called a half-open interval.

The fact Function

- The factorial of a number n (which is usually written as n! in mathematics) is defined to be the product of the integers from 1 up to n. Thus, 5! is equal to 120, which is 1*2*3*4*5.
- The following function definition uses a for loop to compute the factorial function:

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Iterating over Sequences

- The for statement is also used to iterate over the elements of a sequence of values.
- Python supports many different kinds of sequences that you will learn about later in the semester. The only sequence type you’ve seen so far is the string type, which represents a sequence of characters.
- The following function returns the number of times the character a appears in the string s:

```
def countOccurrences(c, s):
    count = 0
    for ch in s:
        if c == ch:
            count += 1
    return count
```