Chapter 4. Choose with if

If you can keep your head when all about you
Are losing theirs and blaming it on you, …

—Rudyard Kipling, If—

In the previous chapters, you've seen many examples of data but haven't done much with them. Most of the code examples used the interactive interpreter and were short. In this chapter, you learn how to structure Python *code*, not just data.

Many computer languages use characters such as curly braces (`{` and `}`) or keywords such as `begin` and `end` to mark off sections of code. In those languages, it's good practice to use consistent indentation to make your program more readable for yourself and others. There are even tools to make your code line up nicely.

When he was designing the language that became Python, Guido van Rossum decided that the indentation itself was enough to define a program's structure, and avoided typing all those parentheses and curly braces. Python is unusual in this use of *white space* to define program structure. It's one of the first aspects that newcomers notice, and it can seem odd to those who have experience with other languages. It turns out that after writing Python for a little while, it feels natural, and you stop noticing it. You even get used to doing more while typing less.

Our initial code examples have been one-liners. Let's first see how to make comments and multiple-line commands.

Comment with

A *comment* is a piece of text in your program that is ignored by the Python interpreter. You might use comments to clarify nearby Python code, make notes to yourself to fix something someday, or for whatever purpose you like. You mark a comment by using the # character; everything from that point on to the end of the current line is part of the comment. You'll usually see a comment on a line by itself, as shown here:

```
>>> # 60 sec/min * 60 min/hr * 24 hr/day
>>> seconds_per_day = 86400
```

Or, on the same line as the code it's commenting:

```
>>> seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

The # character has many names: *hash*, *sharp*, *pound*, or the sinister-sounding *octothorpe*.[1] Whatever you call it,[2] its effect lasts only to the end of the line on which it appears.

Python does not have a multiline comment. You need to explicitly begin each comment line or section with a #:

```
>>> # I can say anything here, even if Python doesn't like it,
... # because I'm protected by the awesome
... # octothorpe.
...
>>>
```

However, if it's in a text string, the mighty octothorpe reverts back to its role as a plain old # character:

```
>>> print("No comment: quotes make the # harmless.")
No comment: quotes make the # harmless.
```

Continue Lines with \

Programs are more readable when lines are reasonably short. The recommended (not required) maximum line length is 80 characters. If you can't say everything you want to say in that length, you can use the *continua-*

tion character: `\` (backslash). Just put `\` at the end of a line, and Python will suddenly act as though you're still on the same line.

For example, if I wanted to add the first five digits, I could do it a line at a time:

```
>>> sum = 0
>>> sum += 1
>>> sum += 2
>>> sum += 3
>>> sum += 4
>>> sum
10
```

Or, I could do it in one step, using the continuation character:

```
>>> sum = 1 + \
...       2 + \
...       3 + \
...       4
>>> sum
10
```

If we skipped the backslash in the middle of an expression, we'd get an exception:

```
>>> sum = 1 +
  File "<stdin>", line 1
    sum = 1 +
            ^
SyntaxError: invalid syntax
```

Here's a little trick—if you're in the middle of paired parentheses (or square or curly brackets), Python doesn't squawk about line endings:

```
>>> sum = (
...     1 +
...     2 +
```

```
...     3 +
...     4)
>>>
>>> sum
10
```

You'll also see in Chapter 5 that paired triple quotes let you make multi-line strings.

Compare with if, elif, and else

Now, we finally take our first step into the *code structures* that weave data into programs. Our first example is this tiny Python program that checks the value of the boolean variable `disaster` and prints an appropriate comment:

```
>>> disaster = True
>>> if disaster:
...     print("Woe!")
... else:
...     print("Whee!")
...
Woe!
>>>
```

The `if` and `else` lines are Python *statements* that check whether a condition (here, the value of `disaster`) is a boolean `True` value, or can be evaluated as `True`. Remember, `print()` is Python's built-in *function* to print things, normally to your screen.

NOTE

If you've programmed in other languages, note that you don't need parentheses for the `if` test. For example, don't say something such as `if (disaster == True)` (the equality operator `==` is described in a few paragraphs). You do need the colon (`:`) at the end. If, like me, you forget to type the colon at times, Python will display an error message.

Each `print()` line is indented under its test. I used four spaces to indent each subsection. Although you can use any indentation you like, Python expects you to be consistent with code within a section—the lines need to be indented the same amount, lined up on the left. The recommended style, called *PEP-8*, is to use four spaces. Don't use tabs, or mix tabs and spaces; it messes up the indent count.

We did a number of things here, which I explain more fully as the chapter progresses:

- Assigned the boolean value `True` to the variable named `disaster`
- Performed a *conditional comparison* by using `if` and `else`, executing different code depending on the value of `disaster`
- *Called* the `print()` *function* to print some text

You can have tests within tests, as many levels deep as needed:

```
>>> furry = True
>>> large = True
>>> if furry:
...     if large:
...         print("It's a yeti.")
...     else:
...         print("It's a cat!")
... else:
...     if large:
...         print("It's a whale!")
...     else:
...         print("It's a human. Or a hairless cat.")
...
It's a yeti.
```

In Python, indentation determines how the `if` and `else` sections are paired. Our first test was to check `furry`. Because `furry` is `True`, Python goes to the indented `if large` test. Because we had set `large` to `True`, `if large` is evaluated as `True`, and the following `else` line is ignored. This makes Python run the line indented under `if large:` and print `It's a yeti.`

If there are more than two possibilities to test, use `if` for the first, `elif` (meaning *else if*) for the middle ones, and `else` for the last:

```
>>> color = "mauve"
>>> if color == "red":
...     print("It's a tomato")
... elif color == "green":
...     print("It's a green pepper")
... elif color == "bee purple":
...     print("I don't know what it is, but only bees can see it")
... else:
...     print("I've never heard of the color", color)
...
I've never heard of the color mauve
```

In the preceding example, we tested for equality by using the `==` operator. Here are Python's *comparison operators*:

equality	`==`
inequality	`!=`
less than	`<`
less than or equal	`<=`
greater than	`>`
greater than or equal	`>=`

These return the boolean values `True` or `False`. Let's see how these all work, but first, assign a value to `x`:

```
>>> x = 7
```

Now, let's try some tests:

```
>>> x == 5
False
>>> x == 7
True
>>> 5 < x
True
>>> x < 10
True
```

Note that two equals signs (`==`) are used to *test equality*; remember, a single equals sign (`=`) is what you use to assign a value to a variable.

If you need to make multiple comparisons at the same time, you use the *logical* (or *boolean*) *operators* `and` , `or` , and `not` to determine the final boolean result.

Logical operators have lower *precedence* than the chunks of code that they're comparing. This means that the chunks are calculated first, and then compared. In this example, because we set `x` to `7` , `5 < x` is calculated to be `True` and `x < 10` is also `True` , so we finally end up with `True and True` :

```
>>> 5 < x and x < 10
True
```

As "Precedence" points out, the easiest way to avoid confusion about precedence is to add parentheses:

```
>>> (5 < x) and (x < 10)
True
```

Here are some other tests:

```
>>> 5 < x or x < 10
True
>>> 5 < x and x > 10
False
```

```
>>> 5 < x and not x > 10
True
```

If you're `and` -ing multiple comparisons with one variable, Python lets you do this:

```
>>> 5 < x < 10
True
```

It's the same as `5 < x and x < 10`. You can also write longer comparisons:

```
>>> 5 < x < 10 < 999
True
```

What Is True?

What if the element we're checking isn't a boolean? What does Python consider `True` and `False` ?

A `false` value doesn't necessarily need to explicitly be a boolean `False` . For example, these are all considered `False` :

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

Anything else is considered `True`. Python programs use these definitions of “truthiness” and “falsiness” to check for empty data structures as well as `False` conditions:

```
>>> some_list = []
>>> if some_list:
...     print("There's something in here")
... else:
...     print("Hey, it's empty!")
...
Hey, it's empty!
```

If what you’re testing is an expression rather than a simple variable, Python evaluates the expression and returns a boolean result. So, if you type:

```
if color == "red":
```

Python evaluates `color == "red"`. In our earlier example, we assigned the string `"mauve"` to `color`, so `color == "red"` is `False`, and Python moves on to the next test:

```
elif color == "green":
```

Do Multiple Comparisons with in

Suppose that you have a letter and want to know whether it's a vowel. One way would be to write a long `if` statement:

```
>>> letter = 'o'
>>> if letter == 'a' or letter == 'e' or letter == 'i' \
...     or letter == 'o' or letter == 'u':
...     print(letter, 'is a vowel')
... else:
...     print(letter, 'is not a vowel')
...
o is a vowel
>>>
```

Whenever you need to make a lot of comparisons like that, separated by `or`, use Python's *membership operator* `in`, instead. Here's how to check vowel-ness more Pythonically, using `in` with a string made of vowel characters:

```
>>> vowels = 'aeiou'
>>> letter = 'o'
>>> letter in vowels
True
>>> if letter in vowels:
...     print(letter, 'is a vowel')
...
o is a vowel
```

Here's a preview of how to use `in` with some data types that you'll read about in detail in the next few chapters:

```
>>> letter = 'o'
>>> vowel_set = {'a', 'e', 'i', 'o', 'u'}
>>> letter in vowel_set
True
>>> vowel_list = ['a', 'e', 'i', 'o', 'u']
>>> letter in vowel_list
True
>>> vowel_tuple = ('a', 'e', 'i', 'o', 'u')
>>> letter in vowel_tuple
True
>>> vowel_dict = {'a': 'apple', 'e': 'elephant',
...               'i': 'impala', 'o': 'ocelot', 'u': 'unicorn'}
>>> letter in vowel_dict
True
>>> vowel_string = "aeiou"
>>> letter in vowel_string
True
```

For the dictionary, `in` looks at the keys (the lefthand side of the `:`) instead of their values.

New: I Am the Walrus

Arriving in Python 3.8 is the *walrus operator*, which looks like this:

```
name := expression
```

See the walrus? (Like a smiley, but tuskier.)

Normally, an assignment and test take two steps:

```
>>> tweet_limit = 280
>>> tweet_string = "Blah" * 50
>>> diff = tweet_limit - len(tweet_string)
>>> if diff >= 0:
...     print("A fitting tweet")
... else:
...     print("Went over by", abs(diff))
```

```
...
A fitting tweet
```

With our new tusk power (aka assignment expressions) we can combine these into one step:

```
>>> tweet_limit = 280
>>> tweet_string = "Blah" * 50
>>> if diff := tweet_limit - len(tweet_string) >= 0:
...     print("A fitting tweet")
... else:
...     print("Went over by", abs(diff))
...
A fitting tweet
```

The walrus also gets on swimmingly with `for` and `while`, which we look at in Chapter 6.

Coming Up

Play with strings, and meet interesting characters.

Things to Do

4.1 Choose a number between 1 and 10 and assign it to the variable `secret`. Then, select another number between 1 and 10 and assign it to the variable `guess`. Next, write the conditional tests (`if`, `else`, and `elif`) to print the string `'too low'` if `guess` is less than `secret`, `'too high'` if greater than `secret`, and `'just right'` if equal to `secret`.

4.2 Assign `True` or `False` to the variables `small` and `green`. Write some `if` / `else` statements to print which of these matches those choices: cherry, pea, watermelon, pumpkin.

1 Like that eight-legged green *thing* that's *right behind you*!

Please
don’t
call
it.
It
might
come
back.