General

Conditional expressions

You can use a conditional statement, if you want the value that must be assigned to a variable to depend on a certain condition. Say, for example, that you work with the variable `x` that refers to the length of a person (in centimeters), and that you want to assign the value `'large'` to the variable `length` if $x > 185$, and the value `'small'` otherwise. Using a conditional statement, this can be done in the following way.

```
>>> x = 100
>>> if x > 185:
...     length = 'large'
... else:
...     length = 'small'
...
>>> length
'large'

>>> x = 190
>>> if x > 185:
...     length = 'large'
... else:
...     length = 'small'
...
>>> length
'large'
```

The same result can be obtained using an `if-else`-expression (also called a *conditional expression*). In contrast with a conditional statement, this is an expression (that evaluates into a value) and not a statement. As a result, you may use conditional expressions in the right-hand side of an assignment statement. Using conditional expressions, the above interactive session can be written much shorter.

```
>>> x = 100
>>> length = 'large' if x > 185 else 'small'
>>> length
'small'

>>> x = 190
>>> length = 'large' if x > 185 else 'small'
>>> length
'large'
```

Testing if the value of a variable belongs to a fixed set of options

The following conditional statements have a logical error in checking whether or not it is weekend or a working day on a given weekday. Actually, the condition will always be `True`, suggesting that it is forever weekend.

```
>>> weekday = 'sunday'
>>> 'weekend' if (weekday == 'saturday' or 'sunday') else 'working day'
'weekend'

>>> weekday = 'monday'
>>> 'weekend' if (weekday == 'saturday' or 'sunday') else 'working day'
'weekend'
```

The reason things go wrong is that the condition is composed out of two smaller conditions, being `weekday == 'saturday'` and `'sunday'`. The second condition is always true, since any string is `True` except for the empty string which is `False`. The correct way of formulating the composed condition is

```
>>> weekday = 'sunday'
>>> 'weekend' if (weekday == 'saturday' or weekday == 'sunday') else 'working day'
'weekend'

>>> weekday = 'monday'
>>> 'weekend' if (weekday == 'saturday' or weekday == 'sunday') else 'working day'
'working day'
```

Note the repetition of the variable name in the condition.

Avoid multiple `print` statements

It is always a good idea to avoid excessive use of `print` statements, in order not to clutter your source code. As such, it becomes much easier to track what exactly will be printed and how the output is formatted.

Say, for example, that you have the following source code

```
>>> x = 3
>>> if x < 5:
...     print('less than 5')
... else:
...     print('more than 5')
...
'less than 5'
```

It is better to rewrite this source code as

```
>>> x = 3
>>> if x < 5:
...    result = 'less'
... else:
...    result = 'more'
...
>>> print('f{result} than 5')
'less than 5'
```

Runway

Add leading zeros using string method `format`

If you want to convert an integer to fixed-length string with leading zeros, you can use the string method `format` and an accompanying *format specifier*. Format specifiers are always put in between the pair of brackets used as a placeholder in the template string. A format specifier always starts with a colon (`:`).

In particular, to convert an integer into a fixed-length string with leading zero, you use the format specifier `:0nd`. In this format specifier, the `0` indicates leading zeros need to be added in case the string would otherwise be shorter than the target length, the letter `d` indicates that the value must be formatted as a digit (number) and the number `n` indicates the target length of the string. If the number you want to convert to a string is longer than the target length, the entire number is converted into a string. In this case, the resulting string will thus be longer than the target length.

```
>>> f'{2}'
'2'
>>> f'{2:02d}'
'02'
>>> f'{34:02d}'
'34'
>>> f'{567:02d}'
'567'
>>> f'{89:06d}'
'000089'
```

There are other ways to convert a number into a fixed-length string with leading zeros. One option is to use the string method `zfill`. You can also compute the difference between the actual length and the target length, to determine how many leading zeros must be prepended. Or you can use a `while` loop to prepend leading zeros until the target length is reached.

```
>>> target = 3
>>> number = str(2)
>>> number.zfill(target)
'002'
>>> leading_zeros = target - len(number)
>>> leading_zeros
2
>>> '0' * leading_zeros + number
'002'
>>> while len(number) < target:
...     number = '0' + number
...
>>> number
'002'
```

Digit work

Check if a number is between two limits

In Python you can check if the value (of a variable) is within a certain interval by using a composite Boolean expression. For example, if you want to check whether or not the number `x` is in the interval $]a, b[$, you can use the following Boolean expression

```
>>> a < x and x < b
```

In mathematics, this condition would be written as $a < x < b$, and Python can use the same kind of shorthand notation.

```
>>> a < x < b:
```

This clearly shows that Guido Van Rossum, who invented the Python programming language, was a trained mathematician. Equally, the condition $a \leq x \leq b$ where the limits are included in the interval can be written using the same shorthand notation in Python.

```
>>> a <= x <= b:
```

Python issues a runtime error if you try to fetch the value of an object referenced by a variable that has not been defined earlier in your source code (not object has been assigned to this variable). This happens for example in the following code fragment

```
>>> x = 4
>>> if x < 3:
...     var = 'ok'
...
>>> print(var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
```

where the `print` statement issues a runtime error that mentions the fact that the variable `var` could not be found in the namespace. If you carefully examine the code fragment, you'll see that the variable `var` is not assigned a value in case the value of `x` is larger than 3 (which is the case in this example).

Finding mates

Assignment vs. equality test

In Python the syntax of an assignment statement uses a single equal sign, where an equality test (check whether two objects have the same value) uses two successive equal signs. To check if the value of the variable `x` equals the integer 2, you write

```
>>> if x == 2:    # correct
...     pass
```

and not

```
>>> if x = 2:     # wrong
  File "<myscript.py>", line 1
    if x = 2:
         ^
SyntaxError: invalid syntax
```