Teach Python 3: Quirks and Features

In Python, most student coding errors will be exceptions or logic errors rather than syntax errors. As a teacher, the more you understand about the quirks and features of Python, the better equipped you will be to help students chase down the source of these errors. This document is an assortment of things I wish I had understood better when I taught Python for the first time.

Below is my list of quirks and/or features that I think you should know about, with a handy emoticon that sums up my opinion on each one ☺. See the numbered sections that follow for more information on each.

1. ☺ All named entities (i.e. functions, modules, variables, etc.) are objects.
2. ☺ Functions support default arguments and can be called with keyword parameters.
3. ☺ You can return multiple values from a function and assign them to multiple variables.
4. 😐 Functions always return a value, whether you tell them to or not.
5. ☹ `if "argh": ...` is legal (and other oddities)
6. ☹ `name = "sam" or "scott"` is legal, but doesn't do what you would expect (and still other oddities)
7. ☹ `1 < x < 10` is legal, but never returns `True` (and even more oddities).
8. ☹ The `while` statement can have an `else` clause.

1. All Named Entities are Objects

In Python, all named entities (i.e. ints, floats, functions, classes, modules, etc.) are objects. They can be assigned to other names, passed as an arguments to functions, and returned from functions. Any name can be global to a module or local to a function.

This can be really useful in the hands of a competent programmer. For example, the sorted function allows you to the option of passing a `key` function as a keyword argument to perform the comparisons. However, this feature creates a number of traps that novice programmers may fall into.

Redefining important names

Here's some example code:

```
int = 4 # this is ok
int2 = 5
x = int(input("enter a number")) # program crashes here
```

In this example, the student has used the name `int` for an integer variable. But this is a pre-defined name for a class which is often used for type conversion. But now `int` holds an integer instead, so the code crashes later when the student tries to use the name `int` as a conversion function.

This is also why importing all the names from a module directly into the namespace (e.g. `from math import *`) is frowned upon. You never know when you may accidentally redefine an important name, such as the name of a constant or helper function that many of the other functions in the module rely upon.

Functions within functions

Because function names are variable names, it follows that just as you can have local variables, you can have local functions as well. That is to say, you can nest a function definition within another function definition.

This can be really useful if you know what you're doing. But it can also lead to errors if a student gets the indentation of their program wrong. An example is below – can you find at least two ways to fix it?

```
def spam():
    print("spam")
    def eggs():       # indentation error
        print("eggs")
spam()
eggs()                # program crashes here
```

2. Default Argument Values and Keyword Arguments

This is cool: Python allows you to specify default values for parameters.

```
def test(x, y = 5):
    print('You called the test function!')
    return x < y
```

In the example above, the test function can be called with 1 or 2 parameters. If it's called with only 1 parameter, the parameter y will get 5 as a default.

Also cool: Python also allows you to use the names of parameters as keywords when you call the function. For example:

`test(2, 6)` assigns 2 to `x` and 6 to `y`.

`test(y = 2, x = 6)` assigns 2 to `y` and 6 to `x`.

These arguments are called "keyword arguments".

Default values and keyword arguments are heavily used by Python's built-in functions such as print and sorted:

`print("hi there")` vs. `print("hi there", end="")`

`a = sorted(a)` vs. `a = sorted(a, reverse=True)`

3. Tuple Assignment

This is kind of cool… You can assign to multiple variables at once, like this:

```
x, y, z = -1, "hi", False
```

Even better, you can also return multiple values from a function, like this:

```
def name_rank_sn():
    return "Sam Scott", "Professor", 12345
```

If a function returns more than one value, you can receive them like this:

```
name, rank, sn = name_rank_sn()
```

This is handy because even before you have taught lists or objects, you can help students to avoid global variables by allowing them to make use of this "multiple return" feature.

In reality, what's happening involves the `tuple` data type. A `tuple` is an immutable `list` which can be written with round brackets. But Python syntax also allows you to leave off the brackets in many contexts. Type these into the shell and the result is the same in each case:

```
>>> (1,2,3)
>>> 1,2,3
```

So the `name_rank_sn` function is returning a tuple, and the assignment works because Python allows you to map an entire `list` or `tuple` of values to an equal length `list` or `tuple` of variable names.

```
a = 1,2,3         # assigns tuple (1,2,3) to a
a,b,c = 1,2,3     # assigns the members of the tuple on the right
                  # to the members of the tuple on the left.
```

4. The `None` Value

Every function always returns a value, whether the code makes it explicit or not. If a function ends without encountering a return statement, it will return the special value `None`, which has the type `NoneType`.

The function below will return `False` if `n` is odd and `None` otherwise.

```
def iseven(n):
    if n % 2 == 1:
        return False
```

`None` has the odd property that if an expression in the shell evaluates to `None`, nothing will not be printed.

So if you type `iseven(4)` into the shell, you will get no response at all. On the other hand, if you type `print(iseven(4))`, the response will be None.

On the other hand, if you use `iseven(n)` in an `if` or `while` statement, the code will act as if the answer is always `False`. To understand why that is, read the next section.

5. If and While With Non-Boolean Expressions

`If` and `while` statements do not require a value of type Boolean. If necessary, any value can be mapped to `False` if it's `0`, the empty string, the empty list or the value `None`; otherwise it will be mapped to `True`.

So the code below will never throw an exception. But if you're not careful about types, it might not do what you expected:

```
if x:
    print("x!")
```

6. The Semantics of `and`, `or`, and `not`

A common mistake students often make is to type `x == 5 or 6` instead of `x == 5 or x == 6`. In Python, this is not a syntax error. Instead, it evaluates to `True` if `x` is 5 and evaluates to `6` if it is not.

Similarly to `if` and `while`, `and`, `or` and `not` do not require Boolean operands. So you can type things like `not "fdsa"`, `not None` and `not int` and get back `True` or `False` (can you predict which you will get in each case?)

On top of that, the semantics of `and` and `or` are unusual.

Semantics of "`x and y`": If x maps to `True`, the result is `x`. Otherwise it's `y`.

Semantics of "`x or y`": If x maps to `False`, the result is `y`. Otherwise it's `x`.

So when `x` and y are both Boolean, `and` and `or` behave like the standard Boolean operators. When one or both of `x` and `y` are not Boolean, they behave differently.

Why did the Python gods allow this? Maybe because it enables things like this:

```
print("Enter a name or hit enter for the default.")
x=input() or "Bilbo Baggins"
print("Your character's name is", x)
```

But this kind of coding is not really very Pythonesque (see "The Zen of Python").

7. Weak Typing and the Logical Operators

Novice programmers sometimes try to type things like `1 < x < 10`. This is a syntax error in many languages. In Python, it's legal and will usually evaluate to `True`.

Python is very strongly typed in some ways (e.g. `"the answer is " + 5` is not allowed). But when it comes to logical operators, it's not.

The == and != operators allow any types to be compared. If the operands are incompatible types (e.g. `str` and `int`) the result will always be false. If they are compatible (e.g. `int` and `float`) then the result will usually make sense.

The <, >, <= and >= operators will throw an exception if the operands are incompatible types.

The Boolean values True and False map to 1 and 0 respectively. So `True == 1` is `True` and `True == 10` is False. In the expression `1 < x < 10`, if `1 < x` is `True`, this becomes `True < 10`, which is `True`. If `1 < x` is `false`, it becomes `False < 10`, which is also `True`.

8. Else as a Completion Clause

This is allowed in Python:

```
x = 1
while x < 10:
    print(x)
    x += 1
else:
    print("x is 10 or more")
```

In this example, the student has mistakenly attached an `else` clause to a `while` loop, as if it was an `if` statement. But in Python, this is legal! The `else` clause will execute when the loop is finished. What's going on here?

In an `if` statement, `else` means what it does in most languages. Elsewhere, `else` is a completion clause. It can be attached to `while`, `for` and `try` statements. It will execute only if the statement it's attached to has "completed". For `while` and `for`, "completed" means they were not terminated with a break statement. For `try`, it means that there was no exception thrown.

Here's an example in which the else class will only sometimes execute.

```
count = 1
while count <= 3:
    print('Enter name', count, 'of 3')
    name = input()
    if name == '':
        break
    count += 1
else:
    print('thank you')
```