Common Gotchas

For the most part, Python aims to be a clean and consistent language that avoids surprises. However, there are a few cases that can be confusing to newcomers.

Some of these cases are intentional but can be potentially surprising. Some could arguably be considered language warts. In general, what follows is a collection of potentially tricky behavior that might seem strange at first glance, but is generally sensible once you’ re aware of the underlying cause for the surprise.

Mutable Default Arguments

Seemingly the *most* common surprise new Python programmers encounter is Python’ s treatment of mutable default arguments in function definitions.

What You Wrote

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

What You Might Have Expected to Happen

```
my_list = append_to(12)
print(my_list)

my_other_list = append_to(42)
print(my_other_list)
```

A new list is created each time the function is called if a second argument isn’ t provided, so that the output is:

```
[12]
[42]
```

What Does Happen

```
[12]
[12, 42]
```

A new list is created *once* when the function is defined, and the same list is used in each successive call.

Python' s default arguments are evaluated *once* when the function is defined, not each time the function is called (like it is in say, Ruby). This means that if you use a mutable default argument and mutate it, you *will* and have mutated that object for all future calls to the function as well.

What You Should Do Instead

Create a new object each time the function is called, by using a default arg to signal that no argument was provided (`None` is often a good choice).

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

Do not forget, you are passing a *list* object as the second argument.

When the Gotcha Isn' t a Gotcha

Sometimes you can specifically "exploit" (read: use as intended) this behavior to maintain state between calls of a function. This is often done when writing a caching function.

Late Binding Closures

Another common source of confusion is the way Python binds its variables in closures (or in the surrounding global scope).

What You Wrote

```
def create_multipliers():
    return [lambda x : i * x for i in range(5)]
```

What You Might Have Expected to Happen

```
for multiplier in create_multipliers():
    print(multiplier(2))
```

A list containing five functions that each have their own closed-over `i` variable that multiplies their argument, producing:

```
0
2
4
6
8
```

What Does Happen

```
8
8
8
8
8
```

Five functions are created; instead all of them just multiply `x` by 4.

Python' s closures are *late binding*. This means that the values of variables used in closures are looked up at the time the inner function is called.

Here, whenever *any* of the returned functions are called, the value of `i` is looked up in the surrounding scope at call time. By then, the loop has completed and `i` is left with its final value of 4.

What's particularly nasty about this gotcha is the seemingly prevalent misinformation that this has something to do with lambdas in Python. Functions created with a `lambda` expression are in no way special, and in fact the same exact behavior is exhibited by just using an ordinary `def`:

```
def create_multipliers():
    multipliers = []

    for i in range(5):
        def multiplier(x):
            return i * x
        multipliers.append(multiplier)

    return multipliers
```

What You Should Do Instead

The most general solution is arguably a bit of a hack. Due to Python's aforementioned behavior concerning evaluating default arguments to functions (see Mutable Default Arguments), you can create a closure that binds immediately to its arguments by using a default arg like so:

```
def create_multipliers():
    return [lambda x, i=i : i * x for i in range(5)]
```

Alternatively, you can use the functools.partial function:

```
from functools import partial
from operator import mul

def create_multipliers():
    return [partial(mul, i) for i in range(5)]
```

When the Gotcha Isn't a Gotcha

Sometimes you want your closures to behave this way. Late binding is good in lots of situations. Looping to create unique functions is unfortunately a case where they can cause hiccups.

Bytecode (.pyc) Files Everywhere!

By default, when executing Python code from files, the Python interpreter will automatically write a bytecode version of that file to disk, e.g. `module.pyc`.

These `.pyc` files should not be checked into your source code repositories.

Theoretically, this behavior is on by default for performance reasons. Without these bytecode files present, Python would re-generate the bytecode every time the file is loaded.

Disabling Bytecode (.pyc) Files

Luckily, the process of generating the bytecode is extremely fast, and isn't something you need to worry about while developing your code.

Those files are annoying, so let's get rid of them!

```
$ export PYTHONDONTWRITEBYTECODE=1
```

With the `$PYTHONDONTWRITEBYTECODE` environment variable set, Python will no longer write these files to disk, and your development environment will remain nice and clean.

I recommend setting this environment variable in your `~/.profile`.

Removing Bytecode (.pyc) Files

Here's nice trick for removing all of these files, if they already exist:

```
$ find . -type f -name "*.py[co]" -delete -or -type d -name "__pycache__" -delete
```

Run that from the root directory of your project, and all `.pyc` files will suddenly vanish. Much better.

Version Control Ignores

If you still need the `.pyc` files for performance reasons, you can always add them to the ignore files of your version control repositories. Popular version control systems have the ability to use wildcards defined in a file to apply special rules.

An ignore file will make sure the matching files don't get checked into the repository. Git uses `.gitignore` while Mercurial uses `.hgignore`.

At the minimum your ignore files should look like this.

```
syntax:glob   # This line is not needed for .gitignore files.
*.py[cod]     # Will match .pyc, .pyo and .pyd files.
__pycache__/  # Exclude the whole folder
```

You may wish to include more files and directories depending on your needs. The next time you commit to the repository, these files will not be included.