How do I manually throw/raise an exception in Python?

Use the most specific Exception constructor that semantically fits your issue.

Be specific in your message, e.g.:

```
raise ValueError('A very specific bad thing happened')
```

Don't do this:

Avoid raising a generic Exception, to catch it, you'll have to catch all other more specific exceptions that subclass it.

Hiding bugs

```
raise Exception('I know Python!') # don't, if you catch, likely to hide bugs.
```

For example:

```
def demo_bad_catch():
    try:
        raise ValueError('represents a hidden bug, do not catch this')
        raise Exception('This is the exception you expect to handle')
    except Exception as error:
        print('caught this error: ' + repr(error))

>>> demo_bad_catch()
caught this error: ValueError('represents a hidden bug, do not catch this',)
```

Won't catch

and more specific catches won't catch the general exception:

```
def demo_no_catch():
    try:
        raise Exception('general exceptions not caught by specific handling')
    except ValueError as e:
        print('we will not catch e')


>>> demo_no_catch()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in demo_no_catch
Exception: general exceptions not caught by specific handling
```

Best Practice:

Instead, use the most specific Exception constructor that semantically fits your issue.

```
raise ValueError('A very specific bad thing happened')
```

which also handily allows an arbitrary number of arguments to be passed to the constructor. This works in

Python 2 and 3.

```
raise ValueError('A very specific bad thing happened', 'foo', 'bar', 'baz')
```

These arguments are accessed by the `args` attribute on the Exception object. For example:

```
try:
    some_code_that_may_raise_our_value_error()
except ValueError as err:
    print(err.args)
```

prints

```
('message', 'foo', 'bar', 'baz')
```

In Python 2.5, an actual `message` attribute was added to BaseException in favor of encouraging users to subclass Exceptions and stop using `args`, but the introduction of `message` and the original deprecation of args has been retracted.

When in `except` clause

When inside an except clause, you might want to, e.g. log that a specific type of error happened, and then reraise. The best way to do this while preserving the stack trace is to use a bare raise statement, e.g.:

```
try:
    do_something_in_app_that_breaks_easily()
except AppError as error:
    logger.error(error)
    raise                 # just this!
    # raise AppError      # Don't do this, you'll lose the stack trace!
```

You can preserve the stacktrace (and error value) with `sys.exc_info()`, but this is way more error prone, prefer to use a bare `raise` to reraise. This is the syntax in Python 2:

```
    raise AppError, error, sys.exc_info()[2] # avoid this.
    # Equivalently, as error *is* the second object:
    raise sys.exc_info()[0], sys.exc_info()[1], sys.exc_info()[2]
```

In Python 3:

```
    raise error.with_traceback(sys.exc_info()[2])
```

Again: avoid manually manipulating tracebacks. It's less efficient and more error prone. And if you're using threading and `sys.exc_info` you may even get the wrong traceback (especially if you're using exception handling for control flow - which I'd personally tend to avoid.)

Python 3, Exception chaining

In Python 3, you can chain Exceptions, which preserve tracebacks:

```
    raise RuntimeError('specific message') from error
```

But beware, this *does* change the error type raised.

Deprecated Methods:

These can easily hide and even get into production code. You want to raise an exception/error, and doing them will raise an error, **but not the one intended!**

Valid in Python 2, but not in Python 3 is the following:

```
raise ValueError, 'message' # Don't do this, it's deprecated!
```

Only valid in much older versions of Python (2.4 and lower), you may still see people raising strings:

```
raise 'message' # really really wrong. don't do this.
```

In all modern versions, this will actually raise a TypeError, because you're not raising a BaseException type. If you're not checking for the right exception and don't have a reviewer that's aware of the issue, it could get into production.

Example Usage:

I raise Exceptions to warn consumers of my API if they're using it incorrectly:

```
def api_func(foo):
    '''foo should be either 'baz' or 'bar'. returns something very useful.'''
    if foo not in _ALLOWED_ARGS:
        raise ValueError('{foo} wrong, use "baz" or "bar"'.format(foo=repr(foo)))
```

Create your own error types when apropos:

> **"I want to make an error on purpose, so that it would go into the except"**

You can create your own error types, if you want to indicate something specific is wrong with your application, just subclass the appropriate point in the exception hierarchy:

```
class MyAppLookupError(LookupError):
    '''raise this when there's a lookup error for my app'''
```

and usage:

```
if important_key not in resource_dict and not ok_to_be_missing:
    raise MyAppLookupError('resource is missing, and that is not ok.')
```