The pass statement

All Versions

Examples

Create a new Exception that can be caught

```
class CompileError(Exception):
    pass
```

Ignore an exception

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Syntax

```
pass
```

Parameters

Remarks

Why would you ever want to tell the interpreter to explicitly do nothing? Python has the syntactical requirement that code blocks (after `if` , `except` , `def` , `class` etc.) cannot be empty.

But sometimes an empty code block is useful in itself. An empty `class` block can definine a new, different class, such as exception that can be caught. An empty `except` block can be the simplest way to express “ask for forgiveness later” if there was nothing to ask for forgiveness for. If an iterator does all the heavy lifting, an empty `for` loop to just run the iterator can be useful.

Therefore, if nothing is supposed to happen in a code block, a `pass` is needed for such a block to not produce an `IndentationError` . Alternatively, any statement (including just a term to be evaluated, like the `Ellipsis` literal `...` or a string, most often a docstring) can be used, but the `pass` makes clear that indeed nothing is supposed to happen, and does not need to be actually evaluated and (at least temporarily) stored in memory. Here is a small annotated collection of the most frequent uses of `pass` that crossed my way – together with some comments on good and bad pratice.

- Ignoring (all or) a certain type of `Exception` (example from `xml`):

  ```
  try:
      self.version = "Expat %d.%d.%d" % expat.version_info
  except AttributeError:
      pass # unknown
  ```

  Note: Ignoring all types of raises, as in the following example from `pandas` , is generally considered bad practice, because it also catches exceptions that should probably be passed on to the caller, e.g. `KeyboardInterrupt` or `SystemExit` (or even `HardwareIsOnFireError` – How do you know you aren't running on a custom box with specific errors defined, which some calling application would want to know about?).

  ```
  try:
      os.unlink(filename_larry)
  except:
      pass
  ```

  Instead using at least `except Error:` or in this case preferably `except OSError:` is considered much better practice. A quick analysis of all python modules I have installed gave me that more than 10% of all `except ...: pass` statements catch all exceptions, so it's still a frequent pattern in python programming.

- Deriving an exception class that does not add new behaviour (e.g. in `scipy`):

```
class CompileError(Exception):
    pass
```

Similarly, classes intended as abstract base class often have an explicit empty `__init__` or other methods that subclasses are supposed to derive. (e.g. `pebl`)

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- Testing that code runs properly for a few test values, without caring about the results (from `mpmath`):

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                         norm=lambda x: norm(x, inf)):
    pass
```

- In class or function definitions, often a docstring is already in place as the *obligatory statement* to be executed as the only thing in the block. In such cases, the block may contain `pass` *in addition* to the docstring in order to say "This is indeed intended to do nothing.", for example in `pebl`:

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- In some cases, `pass` is used as a placeholder to say "This method/class/if-block/... has not been implemented yet, but this will be the place to do it", although I personally prefer the Ellipsis literal `...` (NOTE: python-3 only) in order to strictly differentiate between this and the intentional "no-op" in the previous example. For example, if I write a model in broad strokes, I might write

```
def update_agent(agent):
    ...
```

where others might have

```
def update_agent(agent):
    pass
```

before

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

as a reminder to fill in the `update_agent` function at a later point, but run some tests already to see if the rest of the code behaves as intended. (A third option for this case is `raise NotImplementedError`. This is useful in particular for two cases: Either *"This abstract method should be implemented by every subclass, there is no generic way to define it in this base class"*, or *"This function, with this name, is not yet implemented in this release, but this is what its signature will look like"*)