a temporary workspace to try something out in an isolated environment with any arbitrary version of Python that you want to use. I frequently use the commands you just learned about to try different versions of Python out, install pip packages in a temporary container workspace, and generally give myself a side-effect free sandbox.

If you'd like to learn more about Docker, the documentation on the offical Python docker image page is an excellent interactive introduction.[16] Beyond that, you can peruse the official Docker documentation.[17]

You've now explored the interactive console and launched different versions of Python. In the next section we'll talk about one of the most important tools in your toolbox: pdb for debugging Python programs.

Investigating with pdb Breakpoints

Python includes a built-in debugger called pdb.[18] pdb allows you to halt your program's execution on any given line and use an interactive Python console to inspect your program's state. Since pdb is a built-in part of Python, you can import and use pdb at anytime without running additional external executables other than your program.

To add a breakpoint to your program, pick a target line and then add import pdb; pdb.set_trace() on that line. Up until now, we've been typing and/or pasting code into Python's interactive console, but for this example try running the following example code by saying python3 pdb_example.py:

pdb_example.py

```
Line 1 def get_farm_animals():
     2     farm = ["cow", "pig", "goat"]
     3     return farm
     4
     5 import pdb; pdb.set_trace()  # add breakpoint
     6 animals = ["otter", "seal"]
     7 farm_animals = get_farm_animals()
     8 animals = animals + farm_animals
     9 print(animals)
```

pdb_example.py constructs a list of animals and prints it out. For our purposes, the most important part of pdb_example.py is that we set a breakpoint on line 5 using pdb. (The actual code in the file is just something for us to step through.)

16. https://hub.docker.com/_/python/
17. https://docs.docker.com
18. https://docs.python.org/3/library/pdb.html

After you execute python3 pdb_example.py, you should get dropped into an interactive pdb session:

```
> /code/pdb_example.py(6)<module>()
-> animals = ["otter", "seal"]
(Pdb)
```

The program ran up until line number 6 (which binds animals to the list ["otter", "seal"]). The pdb.set_trace() call caused the program to halt and gave us the interactive pdb prompt shown earlier. Line 6 has not yet been executed, but would be next up as indicated by the arrow (->).

The breakpoint() Shortcut

On Python 3.7 or newer, you can use the new built-in breakpoint() function to set a pdb trace. The default behavior of breakpoint() is to call import pdb; pdb.set_trace(), so using breakpoint() may save you a little typing.[19]

Let's get our bearings a little more by using the pdb command named list to print out the source code near our breakpoint:[20]

```
> /code/pdb_example.py(6)<module>()
-> animals = ["otter", "seal"]
(Pdb) list
  1     def get_farm_animals():
  2         farm = ["cow", "pig", "goat"]
  3         return farm
  4
  5     import pdb; pdb.set_trace()
  6  -> animals = ["otter", "seal"]
  7     farm_animals = get_farm_animals()
  8     animals = animals + farm_animals
  9     print(animals)
[EOF]
(Pdb)
```

Executing the list command prints out the nearby source code (in our case the entire file because pdb_example.py is so short). The arrow has not advanced, so we still have not executed any new code.

The pdb session—effectively—gives you a Python interactive console that allows you to inspect (and even manipulate) your program as it runs. Let's start by trying to inspect the animals variable:

```
(Pdb) animals
```

19. https://docs.python.org/3/whatsnew/3.7.html#pep-553-built-in-breakpoint
20. https://docs.python.org/3/library/pdb.html

```
*** NameError: name 'animals' is not defined
(Pdb)
```

Trying to get information about the animals variable resulted in the pdb session printing a NameError. As we learned earlier, our program stopped just before executing line 6, so the animals variable actually hasn't been assigned to any value yet. In order to advance our program so it executes line 6, we'll need to call the pdb command named next:

```
(Pdb) next
> /code/pdb_example.py(7)<module>()
-> farm_animals = get_farm_animals()
(Pdb) animals
['otter', 'seal']
(Pdb)
```

By running next, you advanced the program one line: line 6 was executed and the arrow now indicates that our current position is just before line 7. By typing animals and then Enter, you were able to output the repr of the animals variable (just as you learned about in code on page 4). pdb shows that animals was indeed bound to a list containing the strings otter and seal.

If we were to call next again, pdb would execute line 7 and farm_animals would be bound to the result of get_farm_animals(). But what if we wanted to investigate the interior of the get_farm_animals function itself? You can use the step command to instruct pdb to stop inside of a called function:

```
(Pdb) step
--Call--
> /code/pdb_example.py(1)get_farm_animals()
-> def get_farm_animals():
(Pdb) step
> /code/pdb_example.py(2)get_farm_animals()
-> farm = ["cow", "pig", "goat"]
(Pdb) list
  1     def get_farm_animals():
  2  ->     farm = ["cow", "pig", "goat"]
  3         return farm
  4
  5     import pdb; pdb.set_trace()
  6     animals = ["otter", "seal"]
  7     farm_animals = get_farm_animals()
  8     animals = animals + farm_animals
  9     print(animals)
[EOF]
(Pdb)
```

Calling step the first time has shows the pdb is currently stopped at line 1 (where the function signature for get_farm_animals is defined). Calling step again moves pdb to just before line 2 that assigns the farm variable to a list of three elements. Calling list confirms our position by indicating that we are inside the get_farm_animals() function about to execute line 2.

Quckily Run the Last pdb Command

If you are in a pdb session, just hitting Enter on a blank (Pdb) prompt line will automatically run the last pdb command again. So, for example, if the last command you ran was next, you can repeatedly hit Enter to continue running next until you arrive at a line you where you would like to stop.

Congratuations, you have now learned the fundamentals of pdb. You are able to set a trace in code, output information about variables currently assigned into your program, advance your program's execution line by line using next, and jump into function calls using step.

Many developers use the print function liberally to debug their code and understand what is going on. pdb takes you one step beyond print debugging by dropping you into an interactive session where you can inspect your program's state without knowing beforehand exactly what variables and objects you want to print.

pdb has many useful commands beyond the few we've explored in this section so far. All of the commands are documented in Python's documentation,[21] but the following table summarizes the commands we've used so far and a select few of my favorites (marked with ✪):

Command	Shortcut	Result
list	l	Print source code near the current line
pp ✪	No shortcut	Pretty print the given expression e.g., 'pp some_variable'. Especially helpful when debugging long lists or nested dictionaries.
next	n	Execute until the next line is reached and then stop again
step	s	Same as next, but stops inside of called functions
where ✪	w	Print a stack trace indicating the current call stack
return	r	Execute until the current function returns and then stop again

21. https://docs.python.org/3/library/pdb.html

Command	Shortcut	Result
continue	c	Resume execution until the next breakpoint (if there are no more breakpoints, the program just continues normally)
quit	q	Quit debugger, program being executed is aborted

You'll notice that many of the commands include one or two letter shortcuts (e.g., n for next) that allow you to run the command without typing out its full name. The shortcuts can help save you some typing when you are in a pdb session.

One of my favorites is the pp ("pretty print") shortcut. pp can help make longer data structures like a list of dictionaries more easily digestible by, for example, using discrete lines for each entry in a long list:

```
(Pdb) object_ids = [{"id": i} for i in range(8)]
(Pdb) object_ids
[{'id': 0}, {'id': 1}, {'id': 2}, {'id': 3}, {'id': 4}, {'id': 5},
{'id': 6}, {'id': 7}]
(Pdb) pp object_ids
[{'id': 0},
 {'id': 1},
 {'id': 2},
 {'id': 3},
 {'id': 4},
 {'id': 5},
 {'id': 6},
 {'id': 7}]
(Pdb)
```

The object_ids variable is a bound to a list containing 8 dictionaries. Running pp object_ids outputs each entry on a separate line making it easier to read.

Some pdb Commands Clash with Python Reserved Words

You may have noticed that some pdb commands clash with Python reserved names. For example, the pdb command list and the pdb command next both clash with Python reserved words: list is a built-in reserved for the list data structure, and next is a built-in function used in iteration.[a,b] (Similarly, the help pdb command clashes with the built-in function named help that you learned about earlier.)

These name clashes become problematic whenever you try to use, for example, list as an actual list instead of the pdb list command that prints source code. For example: if you try to convert range(3) into a list, you'll see a strange error:

```
(Pdb) list(range(3))
*** Error in argument: '(range(3))'
```

```
(Pdb)
```

pdb thinks you've tried to run its list command (and not Python's built-in data structure list). To work around this, you can use one of three alternatives.

1. Escape the pdb command with a ! character:

```
(Pdb) !list(range(3))
[0, 1, 2]
(Pdb)
```

2. Use the print function to avoid the collision in the first place:

```
(Pdb) print(list(range(3)))
[0, 1, 2]
(Pdb)
```

3. Use the pdb pretty print command pp:

```
(Pdb) pp list(range(3))
[0, 1, 2]
(Pdb)
```

a. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
b. https://docs.python.org/3/library/functions.html#next

You've run Python programs via the interactive console, and debugged them using pdb. Next, we'll discuss strategies for reducing debugging by detecting bugs in your source code before you even try to run it.

Detecting Problems Early

Unlike traditionally compiled languages, Python does not require source code to be compiled into machine code before it is run. Instead, Python accepts source code directly and executes it as is. This means that it is possible for you to, for example, run invalid Python source code that never stands a chance of executing or working.

To reduce this risk, many Python projects run static analysis tools to help them validate and verify source code before trying to run it. What are static analysis tools? Static analysis tools don't actually run your code, but read it and inspect it for issues that can be found just by browsing the source code itself. Kind of like a friend peering over your shoulder as you type and letting you know when you've made a mistake before you've even tried to run anything.

In this section we'll talk about two tools for finding and eliminating bugs in your programs ahead of time: flake8 and mypy.