Installing pudb

Install PuDB using the command:

```
pip install pudb
```

If you are using Python 2.5, PuDB version 2013.5.1 is the last version to support that version of Python. urwid 1.1.1 works with Python 2.5, newer versions do not.

Starting the debugger

To start debugging, simply insert:

```
from pudb import set_trace; set_trace()
```

A shorter alternative to this is:

```
import pudb; pu.db
```

Or, if pudb is already imported, just this will suffice:

```
pu.db
```

If you are using Python 3.7 or newer, you can add:

```
# Set breakpoint() in Python to call pudb
export PYTHONBREAKPOINT="pudb.set_trace"
```

in your `~/.bashrc`. Then use:

```
breakpoint()
```

to start pudb.

Insert one of these snippets into the piece of code you want to debug, or run the entire script with:

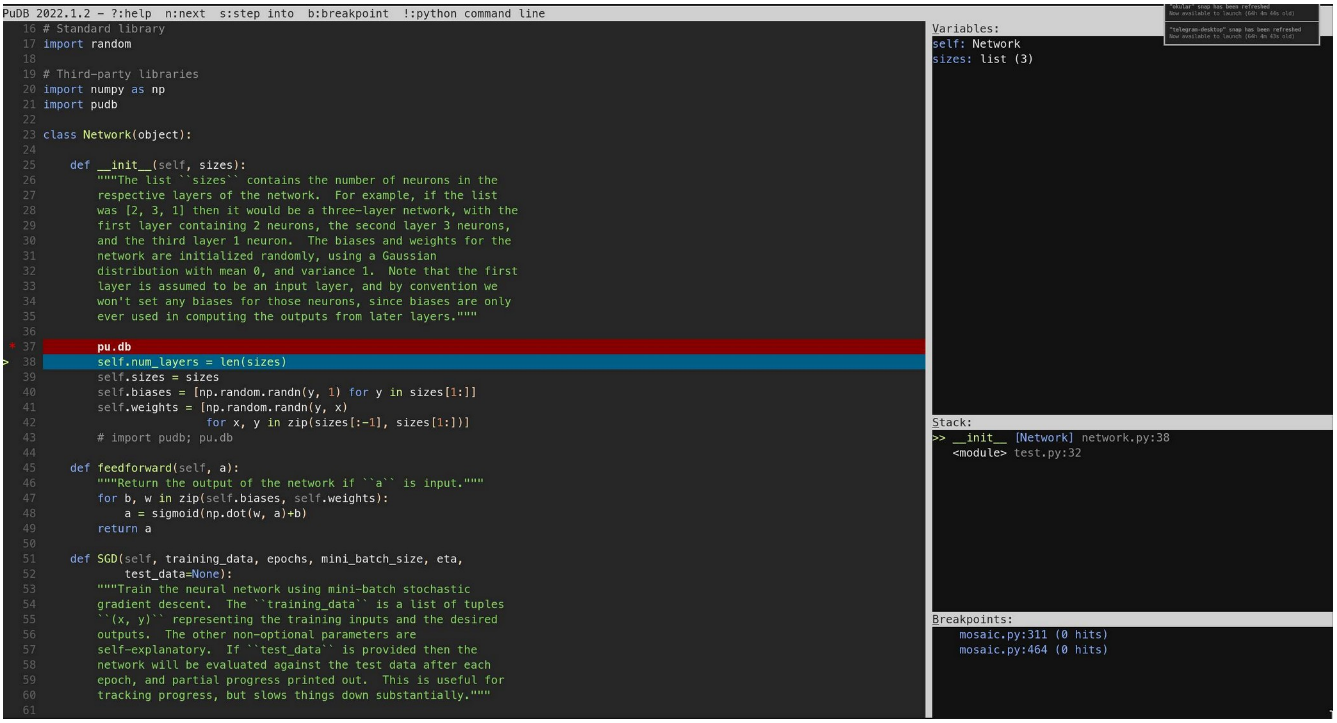
```
python -m pudb my-script.py
```

which is useful if you want to run PuDB in a version of Python other than the one you most recently installed PuDB with.

Using the debugger

Use arrows on your keyboards to navigate through the debugger layout.

- Use up- and down-arrow to move to different lines in a window.
- Use left- and right-arrow to focus on different panes.



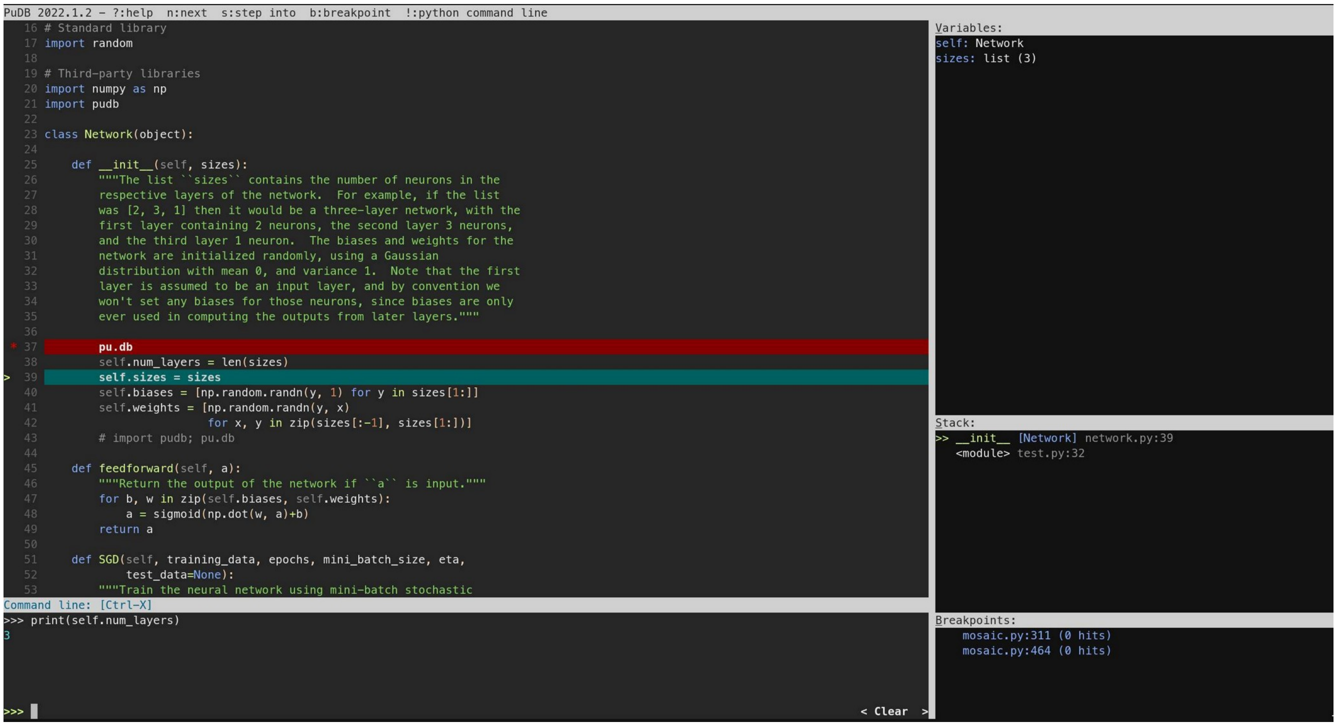
```
PuDB 2022.1.2 - ?:help  n:next  s:step into  b:breakpoint  !:python command line
```

```
"okular" snap has been refreshed
Now available to launch (64h 4m 44s old)

"telegram-desktop" snap has been refreshed
Now available to launch (64h 4m 43s old)
```

```
  # Standard library
  import random

  # Third-party libraries
  import numpy as np
  import pudb

  class Network(object):

      def __init__(self, sizes):
          """The list ``sizes`` contains the number of neurons in the
          respective layers of the network.  For example, if the list
          was [2, 3, 1] then it would be a three-layer network, with the
          first layer containing 2 neurons, the second layer 3 neurons,
          and the third layer 1 neuron.  The biases and weights for the
          network are initialized randomly, using a Gaussian
          distribution with mean 0, and variance 1.  Note that the first
          layer is assumed to be an input layer, and by convention we
          won't set any biases for those neurons, since biases are only
          ever used in computing the outputs from later layers."""

*         pu.db
>         self.num_layers = len(sizes)
          self.sizes = sizes
          self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
          self.weights = [np.random.randn(y, x)
                          for x, y in zip(sizes[:-1], sizes[1:])]
          # import pudb; pu.db

      def feedforward(self, a):
          """Return the output of the network if ``a`` is input."""
          for b, w in zip(self.biases, self.weights):
              a = sigmoid(np.dot(w, a)+b)
          return a

      def SGD(self, training_data, epochs, mini_batch_size, eta,
              test_data=None):
          """Train the neural network using mini-batch stochastic
          gradient descent.  The ``training_data`` is a list of tuples
          ``(x, y)`` representing the training inputs and the desired
          outputs.  The other non-optional parameters are
          self-explanatory.  If ``test_data`` is provided then the
          network will be evaluated against the test data after each
          epoch, and partial progress printed out.  This is useful for
          tracking progress, but slows things down substantially."""
```

```
Variables:
self: Network
sizes: list (3)

Stack:
>> __init__ [Network] network.py:38
   <module> test.py:32

Breakpoints:
    mosaic.py:311 (0 hits)
    mosaic.py:464 (0 hits)
```

Press “Ctrl-X” to bring up/close/set focus to pudb console that can execute python code:



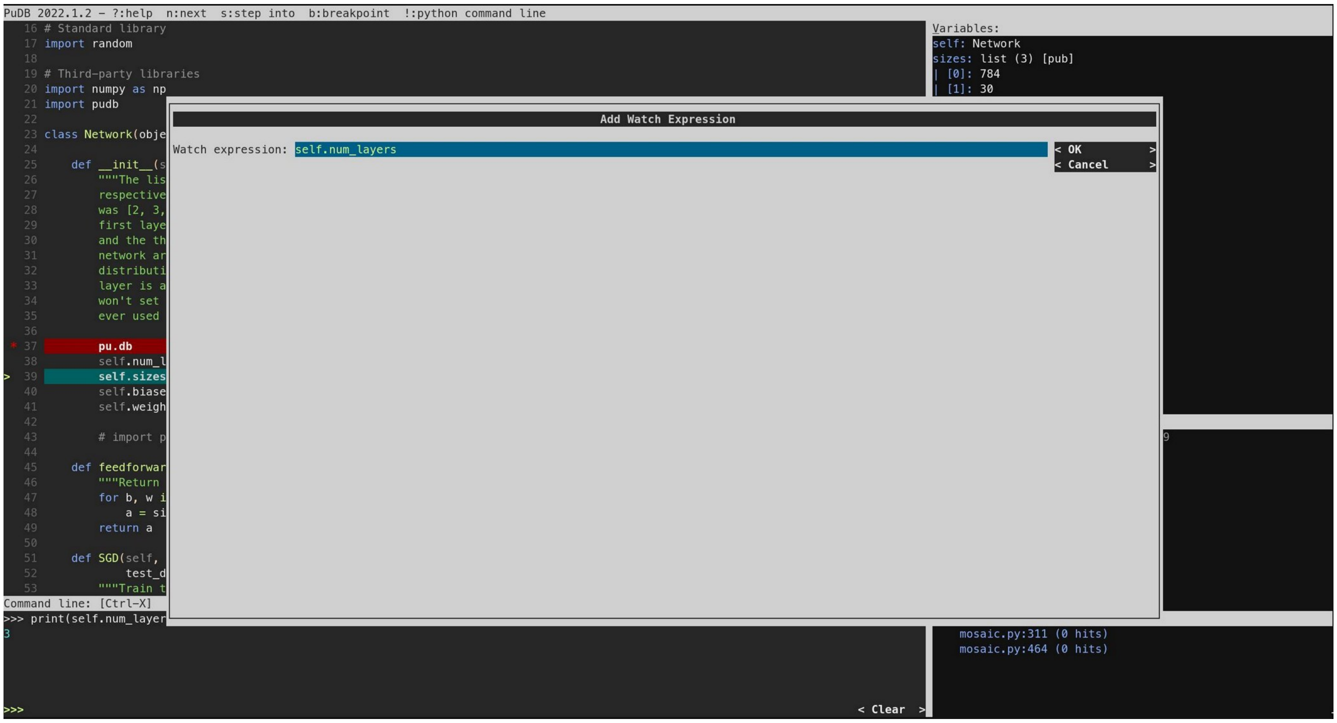
```
PuDB 2022.1.2 - ?:help  n:next  s:step into  b:breakpoint  !:python command line
```

```
  # Standard library
  import random

  # Third-party libraries
  import numpy as np
  import pudb

  class Network(object):

      def __init__(self, sizes):
          """The list ``sizes`` contains the number of neurons in the
          respective layers of the network.  For example, if the list
          was [2, 3, 1] then it would be a three-layer network, with the
          first layer containing 2 neurons, the second layer 3 neurons,
          and the third layer 1 neuron.  The biases and weights for the
          network are initialized randomly, using a Gaussian
          distribution with mean 0, and variance 1.  Note that the first
          layer is assumed to be an input layer, and by convention we
          won't set any biases for those neurons, since biases are only
          ever used in computing the outputs from later layers."""

*         pu.db
          self.num_layers = len(sizes)
>         self.sizes = sizes
          self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
          self.weights = [np.random.randn(y, x)
                          for x, y in zip(sizes[:-1], sizes[1:])]
          # import pudb; pu.db

      def feedforward(self, a):
          """Return the output of the network if ``a`` is input."""
          for b, w in zip(self.biases, self.weights):
              a = sigmoid(np.dot(w, a)+b)
          return a

      def SGD(self, training_data, epochs, mini_batch_size, eta,
              test_data=None):
          """Train the neural network using mini-batch stochastic
```

```
Command line: [Ctrl-X]
>>> print(self.num_layers)
3


>>>                                                        < Clear  >
```

```
Variables:
self: Network
sizes: list (3)

Stack:
>> __init__ [Network] network.py:39
   <module> test.py:32

Breakpoints:
    mosaic.py:311 (0 hits)
    mosaic.py:464 (0 hits)
```

Press “n” on the “Variables” pane to set the value to be run-timely monitored.


PuDB 2022.1.2 - ?:help n:next s:step into b:breakpoint !:python command line
Add Watch Expression
< OK >
< Cancel >
Command line: [Ctrl-X]
< Clear >

```
# Standard library
import random

# Third-party libraries
import numpy as np
import pudb

class Network(obje

    def __init__(s
        """The lis
        respective
        was [2, 3,
        first laye
        and the th
        network ar
        distributi
        layer is a
        won't set
        ever used

        pu.db
        self.num_l
        self.sizes
        self.biase
        self.weigh

        # import p

    def feedforwar
        """Return
        for b, w i
            a = si
        return a

    def SGD(self,
            test_d
        """Train t
```

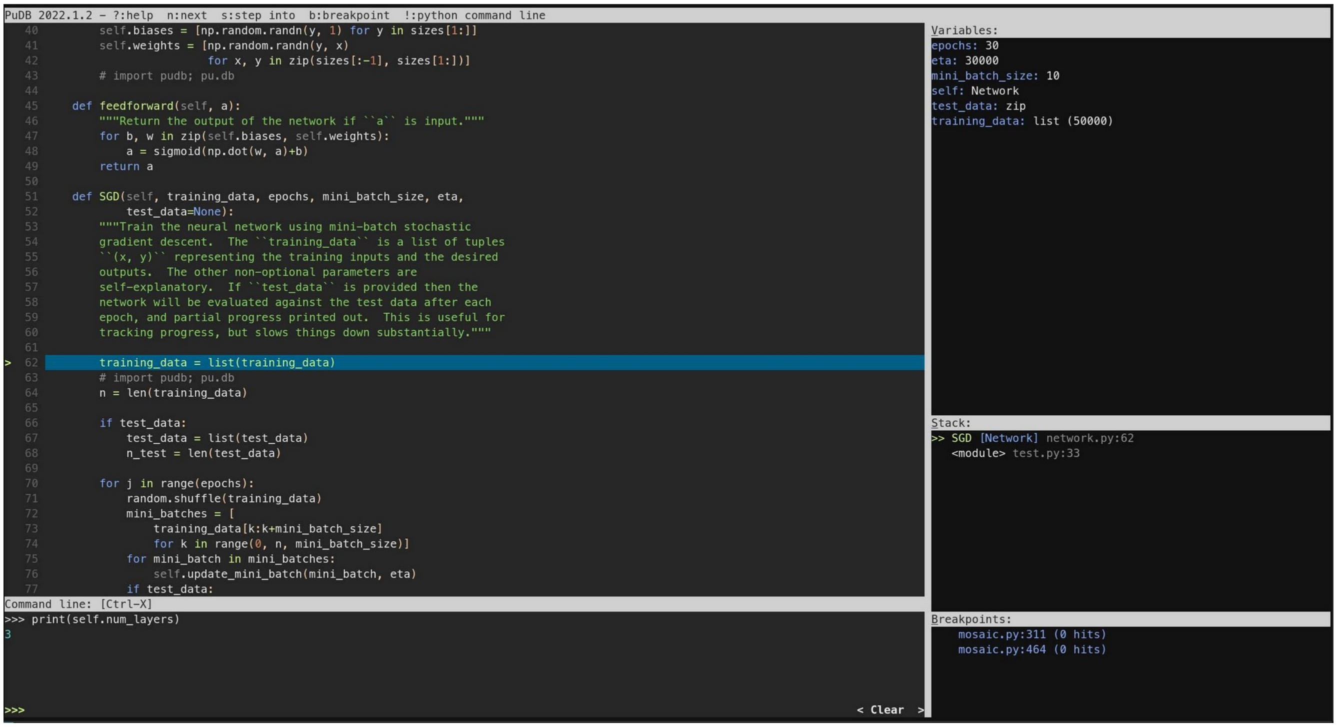
Watch expression: self.num_layers

```
Variables:
self: Network
sizes: list (3) [pub]
| [0]: 784
| [1]: 30
```

```
>>> print(self.num_layer
3
>>>
```

```
mosaic.py:311 (0 hits)
mosaic.py:464 (0 hits)
```

Press “T” to run until a target line is hit.


PuDB 2022.1.2 - ?:help n:next s:step into b:breakpoint !:python command line
Command line: [Ctrl-X]
< Clear >

```
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
        # import pudb; pu.db

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""

>       training_data = list(training_data)
        # import pudb; pu.db
        n = len(training_data)

        if test_data:
            test_data = list(test_data)
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
```

```
Variables:
epochs: 30
eta: 30000
mini_batch_size: 10
self: Network
test_data: zip
training_data: list (50000)
```

```
Stack:
>> SGD [Network] network.py:62
   <module> test.py:33
```

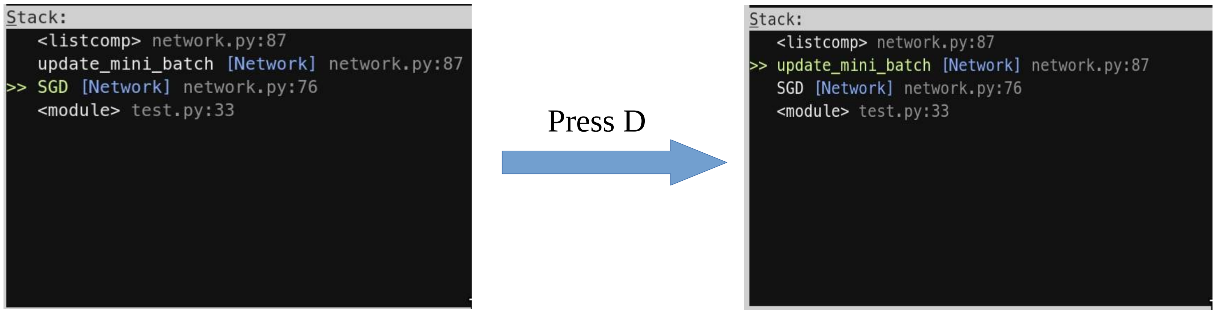
```
>>> print(self.num_layers)
3
>>>
```

```
Breakpoints:
    mosaic.py:311 (0 hits)
    mosaic.py:464 (0 hits)
```

Press “S” to start step-debugging (go “inside” to a function).

Press “D” to travel down the stack (go to the callee). Press “U” to travel up the stack (go to the caller).

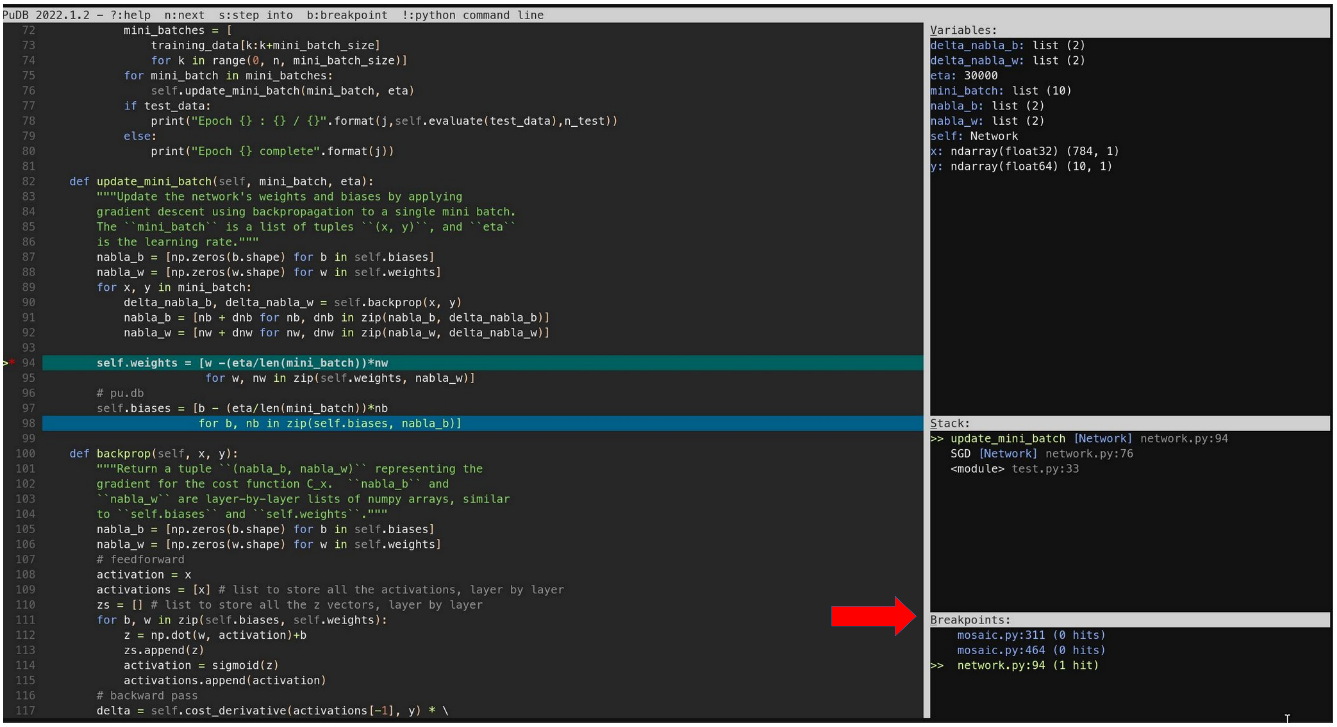
You can always monitor where you are in a program by watching the “Stack” window.



```
Stack:
    <listcomp> network.py:87
    update_mini_batch [Network] network.py:87
>> SGD [Network] network.py:76
    <module> test.py:33
```

```
Stack:
    <listcomp> network.py:87
>> update_mini_batch [Network] network.py:87
    SGD [Network] network.py:76
    <module> test.py:33
```

Press “B” to set breakpoint of a line (you can see the lines with breakpoint activated with a red *. You can also monitor all breakpoints in the Breakpoints window). Then press “C” to continue program execution until the breakpoint.



```
PuDB 2022.1.2 - ?:help  n:next  s:step into  b:breakpoint  !:python command line
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test))
            else:
                print("Epoch {} complete".format(j))

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

        self.weights = [w -(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        # pu.db
        self.biases = [b - (eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
```

```
Variables:
delta_nabla_b: list (2)
delta_nabla_w: list (2)
eta: 30000
mini_batch: list (10)
nabla_b: list (2)
nabla_w: list (2)
self: Network
x: ndarray(float32) (784, 1)
y: ndarray(float64) (10, 1)

Stack:
>> update_mini_batch [Network] network.py:94
   SGD [Network] network.py:76
   <module> test.py:33

Breakpoints:
    mosaic.py:311 (0 hits)
    mosaic.py:464 (0 hits)
>>  network.py:94 (1 hit)
```

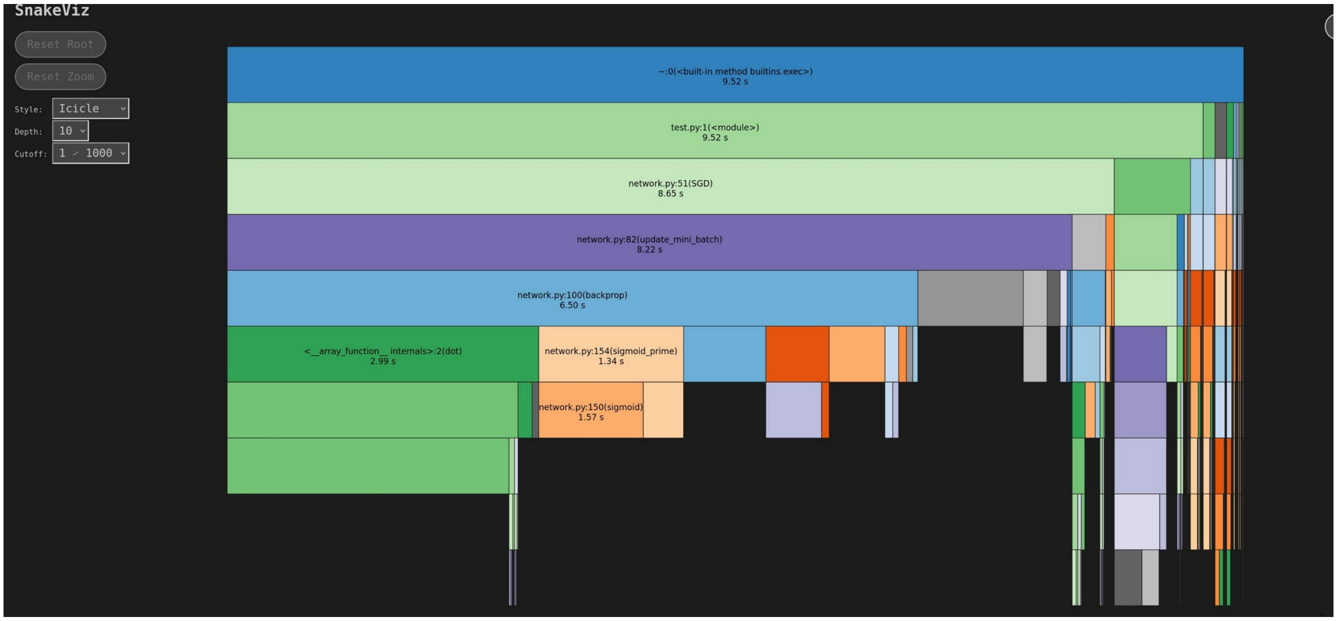
Profiling python code

Python includes a profiler called cProfile. It not only gives the total running time, but also times each function separately, and tells you how many times each function was called, making it easy to determine where you should make optimizations.

python -m cProfile -o test.py.profile test.py

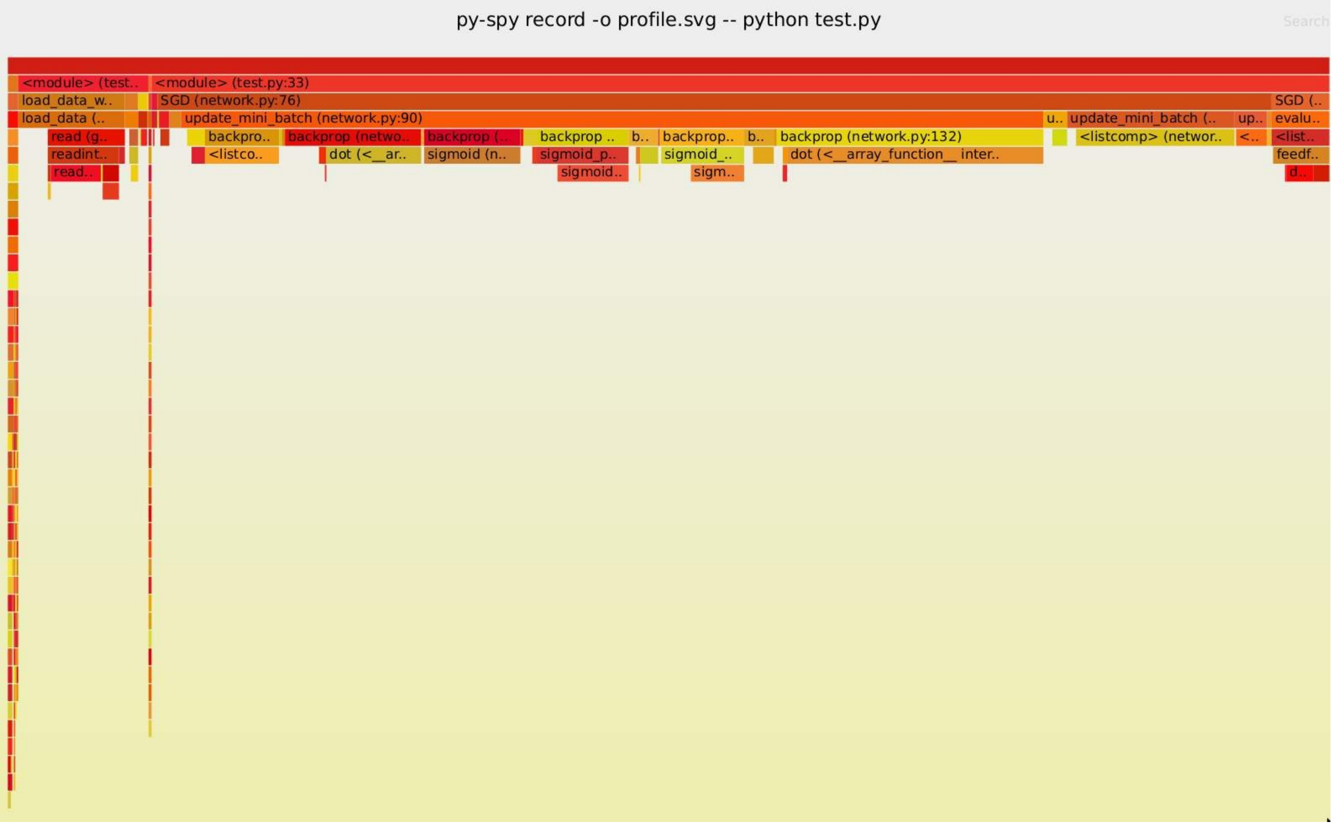
Then you can visualize the results with snakeviz:

- Install snakeviz: pip install snakeviz
- Visualizing the profiling results: snakeviz test.py.profile


SnakeViz
Reset Root
Reset Zoom
Style: Icicle
Depth: 10
Cutoff: 1 ⁄ 1000
~:0(<built-in method builtins.exec>) 9.52 s
test.py:1(<module>) 9.52 s
network.py:51(SGD) 8.65 s
network.py:82(update_mini_batch) 8.22 s
network.py:100(backprop) 6.50 s
<__array_function__ internals>:2(dot) 2.99 s
network.py:154(sigmoid_prime) 1.34 s
network.py:150(sigmoid) 1.57 s

Or you can do the visualization with py-spy:

- Install py-spy: pip install py-spy
- Profiling and visulizing: py-spy record -o profile.svg -- python test.py
- The profiling results will be saved to profile.svg.



```
py-spy record -o profile.svg -- python test.py
```