10 WAYS TO DEBUG PYTHON CODE

Christoph Deil, PyConDE 2019
Slides at https://christophdeil.com

WHAT IS DEBUGGING?

Dictionary

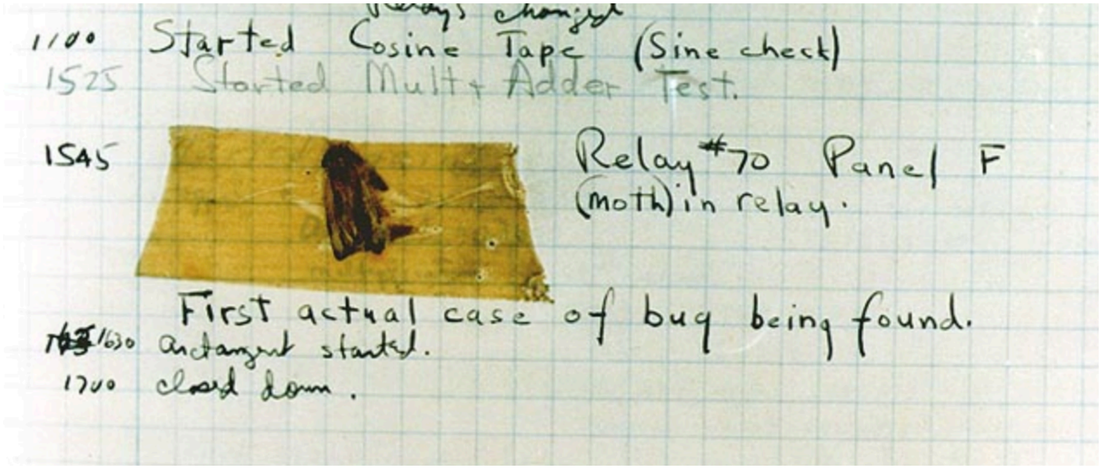
debug | diː'bʌg |

verb (**debugs, debugging, debugged**) *[with object]*

1 identify and remove errors from (computer hardware or software): *games are the worst to debug*.

2 detect and remove concealed microphones from (an area).

3 *North American* remove insects from (something), especially with a pesticide.



Harvard Mark II computer operator log 1947.
Use of term "bug" to describe defects or malfunctions appears in engineering jargon since 1870s
https://en.wikipedia.org/wiki/Software_bug

LEARN DEBUGGING?

- Try to avoid bugs and debugging!
  Write clean & simple code & tests.
- Debugging is often annoying, frustrating, unpredictable how long it will take
- Personal cost: life quality
- Economic cost: 100s billion EUR / year

LEARN DEBUGGING!

- There will be bugs and debugging!
- If you’re a programmer or data scientist, debugging is unavoidable
- Learn and train to be ready and efficient
- Basics are simple.
  Time investment will pay off.

WHY LEARN 10 WAYS?

- Different tasks require different tools:
  - *Code in PyCharm? → Debug in PyCharm!*
  - *Work in Jupyter? → Debug in Jupyter!*
  - *Too many bugs? → Write and debug tests!*
  - *Bad performance → Profiling*
  - *Bugs in production → Logging*
  - …

- Goal of this presentation:
  - *Overview for beginners*
  - *“Learn what to learn”*

10 WAYS TO DEBUG PYTHON CODE — OVERVIEW

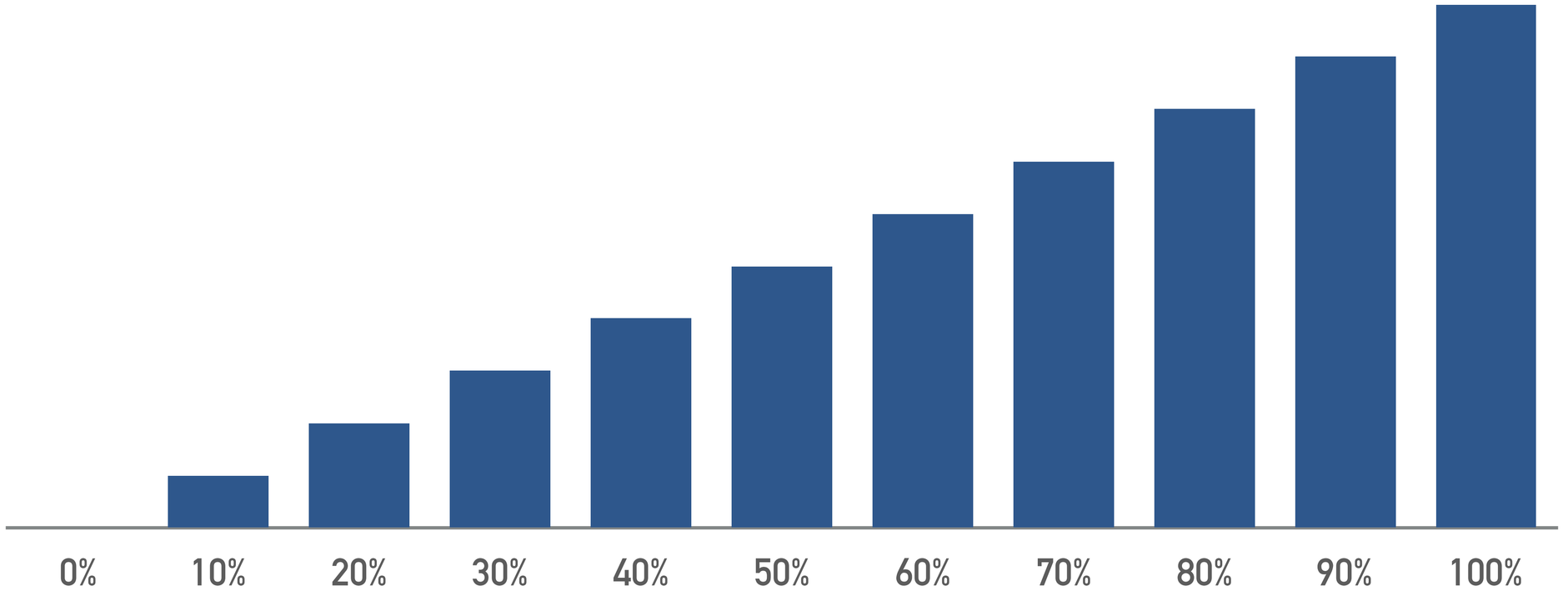
1. Read code
2. Read tracebacks
3. print
4. Python debugger (pdb)
5. IPython & Jupyter
6. PyCharm & VS Code
7. test
8. profile
9. log
10. duck

Many topics not covered, e.g. no concurrency, C extensions, web apps.

DO YOU USE A DEBUGGER?

- Yes, regularly
- Yes, sometimes
- No, almost never

WHAT FRACTION OF YOUR DEVELOPMENT TIME DO YOU SPEND DEBUGGING?



1. READ CODE

point.py

```python
import math

def distance(p1, p2):
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    d2 = dx * dx + dy * dy
    return math.sqrt(d2)

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

analysis.py

```python
from point import Point, distance

p1 = Point(3, 0)
p2 = Point(0, 3)
p2.move(0, 1)
d = distance(p1, p2)
print(d)
```

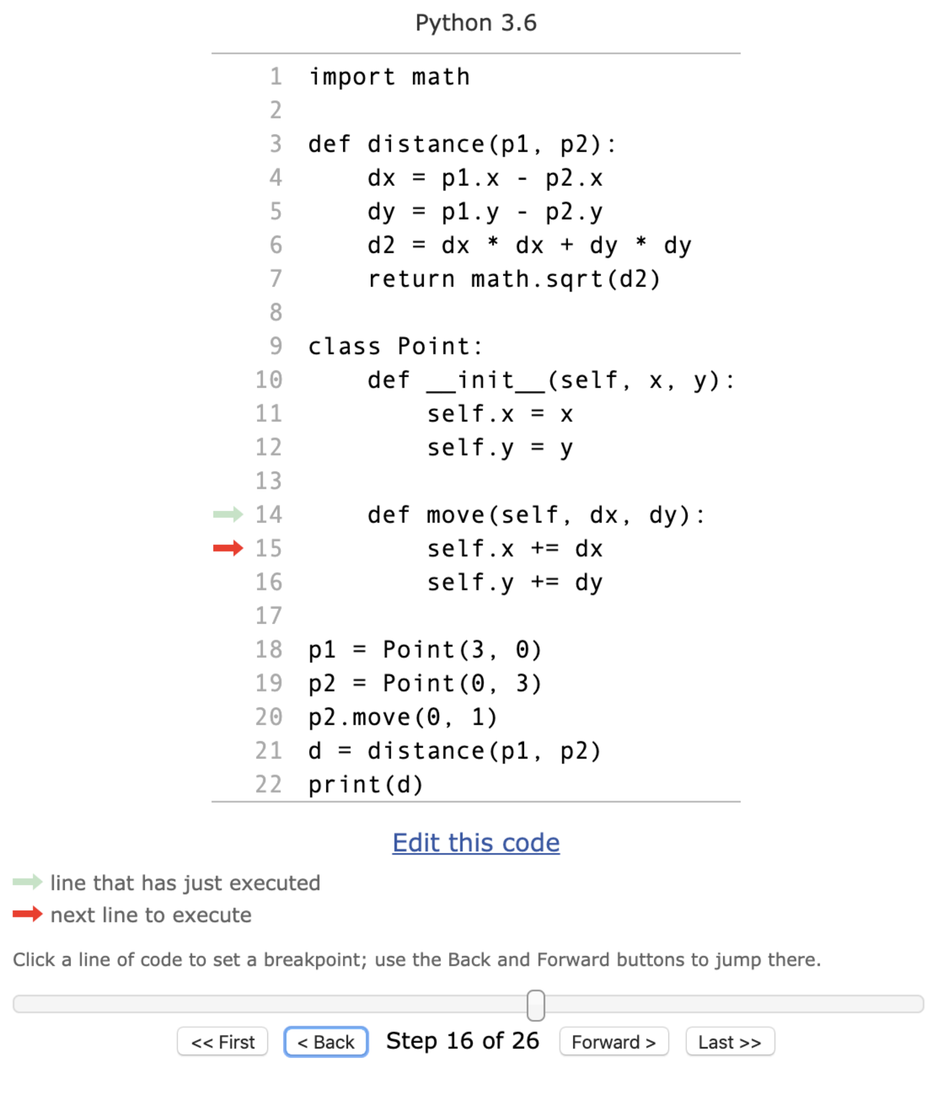
LEARN TO READ PYTHON CODE

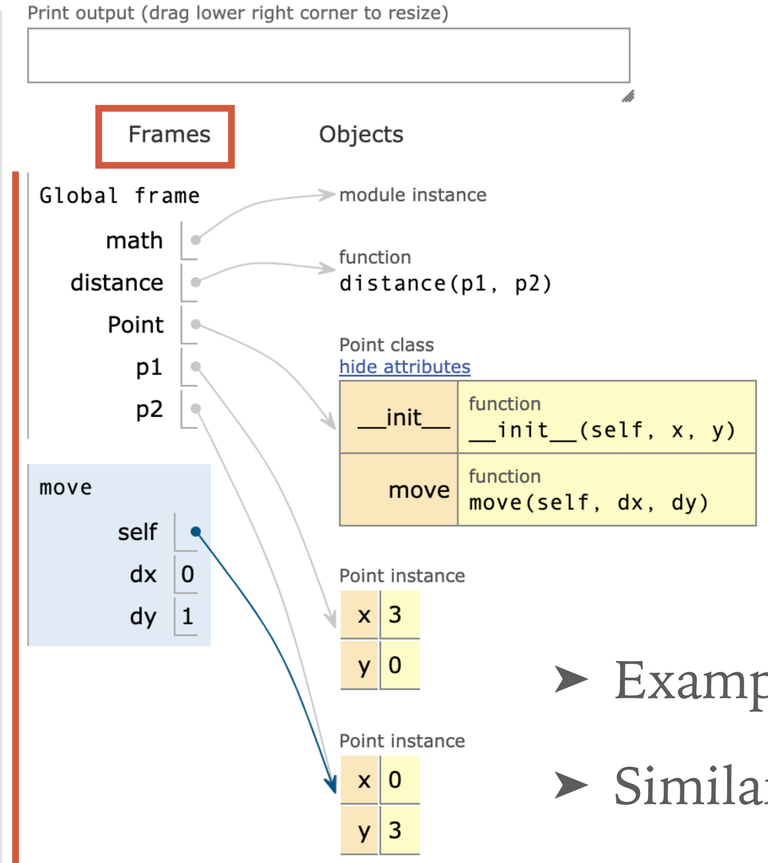
- Need mental model for code execution

```
$ python analysis.py
5.0
```

- Execution:
  - *Mostly top to bottom*
  - *Function calls create stack frames*
  - *Import statements execute other files*
- Everything is an object
  - *"def" → function object*
  - *"class" → class object*
  - *"import" → module object*
- Variables are references to objects

PYTHONTUTOR.COM — DEMO



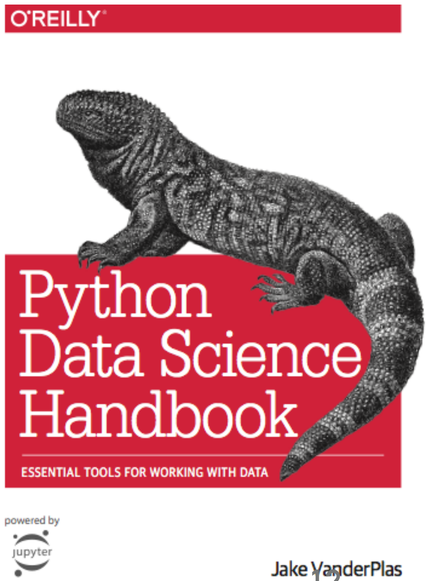


Stack of frames. Function call: add function return: pop

- Example: DEMO LINK
- Similar to visual debuggers:
  - point to next line, step, variables, frames, objects

LEARN TO READ CODE — TIPS

- Have a clear mental model how Python executes code!
  It's the basis of all Python code reading, writing, debugging, …
- Try examples with pythontutor.com or nbtutor or a visual debugger
- Read Python tutorials. Some good free resources:
  - Official Python tutorial
  - Whirlwind tour of Python by Jake VanderPlas
  - Python Data Science Handbook by Jake VanderPlas
  - Python epiphanies by Stuart Williams (YouTube)



2. READ TRACEBACKS

count.py

```python
from pathlib import Path

def count_lines(filename):
    path = Path(filename)
    txt = path.read_text()
    lines = txt.split("\n")
    return len(lines)

n_lines = count_lines("spam.txt")
print(f"Number of lines: {n_lines}")
```

LEARN TO READ TRACEBACKS

- Debugging often starts with an exception and traceback ("function call stack")
- *"Silent bugs" with incorrect output, but no exception, are harder — where to start?*

```
$ python count.py
Traceback (most recent call last):
  File "count.py", line 9, in <module>
    n_lines = count_lines("spam.txt")
  File "count.py", line 5, in count_lines
    txt = path.read_text()
  File "/Users/deil/software/anaconda3/lib/python3.7/pathlib.py", line 1199, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
  File "/Users/deil/software/anaconda3/lib/python3.7/pathlib.py", line 1186, in open
    opener=self._opener)
  File "/Users/deil/software/anaconda3/lib/python3.7/pathlib.py", line 1039, in _opener
    return self._accessor.open(self, flags, mode)
FileNotFoundError: [Errno 2] No such file or directory: 'spam.txt'
```

Stack frame

Stack frame

Read function call stack to see where the error occurred

Often the bug is in "your code" and you can ignore the part from standard libraries

Check exception type and error message first

```
>>> def add(a, b):
... return a + b
  File "<stdin>", line 2
    return a + b
        ^
IndentationError: expected an indented block
```

```
>>> println("Hello world")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'println' is not defined
```

```
>>> conferences = ["pycon" "pydata"]
>>> conferences.copi()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'copi'
```

```
>>> conferences[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> python_skills = {"guide": 8, "christoph": 3}
>>> python_skills["gido"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'gido'
```

```
>>> "spam" === "ham"
  File "<stdin>", line 1
    "spam" === "ham"
           ^
SyntaxError: invalid syntax
```

EXCEPTIONS AND ERRORS

- In Python, “Exception” and “Error” often mean the same thing: instances of a class that derives from BaseException:

```
>>> NameError.__mro__
(<class 'NameError'>, <class 'Exception'>,
<class 'BaseException'>, <class 'object'>)
```

- SyntaxError and IndentationError occur on import, the rest on line execution
- With Python, you’ll get errors all day long. It’s a feature, not a bug!

```
>>> import this
The Zen of Python, by Tim Peters
...
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...
```

exception_chain.py

```python
a, b = 1, 0
try:
    result = a / b
except ZeroDivisionError:
    result = c

print(f"Result: {result}")
```

CHAINED EXCEPTIONS

- Chained exception: two (or more) exceptions and tracebacks
- Occurs when a second exception happens in except part of a try-except statement
- This example: bug in error handling code
- You'll sometimes get this from libraries that use try-except for control flow
- Keep calm and read both.
  (Sometimes you only care about the second one)

```
$ python exception_chain.py
Traceback (most recent call last):
  File "exception_chain.py", line 4, in <module>
    result = a / b
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "exception_chain.py", line 6, in <module>
    result = c
NameError: name 'c' is not defined
```

EXCEPTIONS & TRACEBACKS — TIPS

- Learn the common exception types and common bugs that cause each one.
- Python tutorial on Errors & Exceptions and reference for Built-in Exceptions
- Uncaught exception: Python interpreter prints traceback and exits.
- Learn to read tracebacks and how it connects to your code.
  Function call stacks, last called function where exception occurred at the bottom.
- Often carefully reading the traceback and source code will let you find the bug.
  If not → re-run and use a debugger!

3. PRINT

point.py

```
1   import math
2
3   def distance(p1, p2):
4       dx = p1.x - p2.x
5       dy = p1.y - p2.y
6       d2 = dx * dx + dy * dy
7       return math.sqrt(d2)
8
9   class Point:
10      def __init__(self, x, y):
11          self.x = x
12          self.y = y
13
14      def move(self, dx, dy):
15          self.x += dx
16          self.y += dy
```

print(…)

exception.py

```
1   from point import Point
2
3   def move_it(point):
4       point.move(1, 2)
5
6   def main():
7       p = Point("10", "20")
8       move_it(p)
9
10  main()
```

print(…)

print(…)

PRINT DEBUGGING

- Very common way to debug:
  add “print” in various places
- Where to print which information?
  (usually: a lot of code & files)
- Slow, annoying, error-prone

```
$ python exception.py
Traceback (most recent call last):
  File "exception.py", line 10, in <module>
    main()
  File "exception.py", line 8, in main
    move_it(p)
  File "exception.py", line 4, in move_it
    point.move(1, 2)
  File "/private/tmp/debug/point.py", line 15, in move
    self.x += dx
TypeError: can only concatenate str (not "int") to str
```

PRINT DEBUGGING — TIPS

- Print debugging is slow, annoying and error-prone:
  - add print, run, add another print, re-run, iterate many times.
  - Forget to remove? Edit files in other projects?
- **Don't use print → learn how to use a debugger!**
- *Adding logging.debug to poke around is not much better than adding print.*
  Systematic testing and logging are useful though — see #7 and #9 later.

4. PYTHON DEBUGGER (PDB)


```
$ python -m pdb exception.py
> /private/tmp/debug/exception.py(1)<module>()
-> from point import Point
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb) q
$ _
```

PYTHON DEBUGGER (PDB)

- PDB — feature-full command line debugger in the Python standard library
- Common ways to start it:
  - **python -m pdb myscript.py**
  - Or add one line in code:
    breakpoint() *# since Python 3.7*
    import pdb; pdb.set_trace() # older Python
- Need to learn 5 - 10 commands to use it. Use “h” or “help” to see them.

PDB COMMANDS

- h — help
- q — quit
- p — print (use “pp” to pretty-print)
- ll — “long list” source code in current function or frame
- w — where (print stack trace)
- n — next (“step over”)
- s — step (“step into”)
- c — continue (run to breakpoint or exception or program end)
- b — breakpoint (add or list, use “cl” to remove)
- u — up (in stack frame)
- d — down (in stack frame)

PDB — DEMO

point.py

```python
import math

def distance(p1, p2):
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    d2 = dx * dx + dy * dy
    return math.sqrt(d2)

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

exception.py

```python
from point import Point

def move_it(point):
    point.move(1, 2)

def main():
    p = Point("10", "20")
    move_it(p)

main()
```

```
$ python -m pdb exception.py
> /private/tmp/debug/exception.py(1)<module>()
-> from point import Point
(Pdb) c
Traceback (most recent call last):
  File "/Users/deil/software/anaconda3/lib/python3.7/pdb.py", line 1701, in main
    pdb._runscript(mainpyfile)
  File "/Users/deil/software/anaconda3/lib/python3.7/pdb.py", line 1570, in _runscript
    self.run(statement)
  File "/Users/deil/software/anaconda3/lib/python3.7/bdb.py", line 585, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "/private/tmp/debug/exception.py", line 1, in <module>
    from point import Point
  File "/private/tmp/debug/exception.py", line 8, in main
    move_it(p)
  File "/private/tmp/debug/exception.py", line 4, in move_it
    point.move(1, 2)
  File "/private/tmp/debug/point.py", line 15, in move
    self.x += dx
TypeError: can only concatenate str (not "int") to str
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /private/tmp/debug/point.py(15)move()
-> self.x += dx
(Pdb) p self.x
'10'
(Pdb) p type(self.x)
<class 'str'>
```

PYTHON DEBUGGER (PDB) – TIPS

- Python debugger (PDB) is part of Python standard library, always available
- Command line interface, a bit hard to learn and remember ("h" to print help)
  Suggest you try both PDB and a visual debugger (see later) and see what you like.
- Multiple ways to start PDB: post mortem, step and continue, breakpoints
  Multiple ways so poke around: print, where, list, up, down
- Good resources:
  - Python Debugging With Pdb tutorial by Nathan Jennings on RealPython.com
  - Python module of the week tutorial for pdb by Doug Hellman
  - Python standard library documentation for pdb

5. IPYTHON & JUPYTER

```
$ ipython --no-banner

In [1]: %xmode plain
Exception reporting mode: Plain

In [2]: %run exception.py
Traceback (most recent call last):
  File "/private/tmp/debug/exception.py", line 10, in <module>
    main()
  File "/private/tmp/debug/exception.py", line 8, in main
    move_it(p)
  File "/private/tmp/debug/exception.py", line 4, in move_it
    point.move(1, 2)
  File "/private/tmp/debug/point.py", line 15, in move
    self.x += dx
TypeError: can only concatenate str (not "int") to str


In [3]: %debug
> /private/tmp/debug/point.py(15)move()
     13
     14     def move(self, dx, dy):
---> 15         self.x += dx
     16         self.y += dy
     17

ipdb> type(self.x)
<class 'str'>
ipdb> exit

In [4]: exit
$
```

```
$ ipython -i analysis.py
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
5.0

In [1]: %whos
Variable   Type        Data/Info
------------------------------
Point      type        <class 'point.Point'>
d          float       5.0
distance   function    <function distance at 0x10eb2cd90>
p1         Point       <point.Point object at 0x10eb642e8>
p2         Point       <point.Point object at 0x10eb640f0>
```

IPYTHON

- IPython & Jupyter provide nicer interactive REPL & debugger
- ipdb commands the same as pdb
- Just nicer to use: color, multi-line edit, tab completion, magic commands
- **%run** — run script, -d option
- **%debug** — post-mortem enter **ipdb**
- **%pdb on** — auto-enter ipdb on error
- **%xmode** *(plain, context, verbose, minimal)*
- **ipython -i myscript.py**
- **ipython --pdb**
- **import IPython; IPython.embed()**

```
[1]: def func1(a, b):
         return a / b

     def func2(x):
         a = x
         b = x - 1
         return func1(a, b)

[2]: %xmode minimal

     Exception reporting mode: Minimal

[3]: func2(1)

     ZeroDivisionError: division by zero

[*]: %debug

     > <ipython-input-1-586ccabd0db3>(2)func1()
           1 def func1(a, b):
     ----> 2     return a / b
           3
           4 def func2(x):
           5     a = x

     ipdb>  locals()
     {'a': 1, 'b': 0}
     ipdb>  up
     > <ipython-input-1-586ccabd0db3>(7)func2()
           3
           4 def func2(x):
           5     a = x
           6     b = x - 1
     ----> 7     return func1(a, b)

     ipdb>  p x
     1
     ipdb> q
```

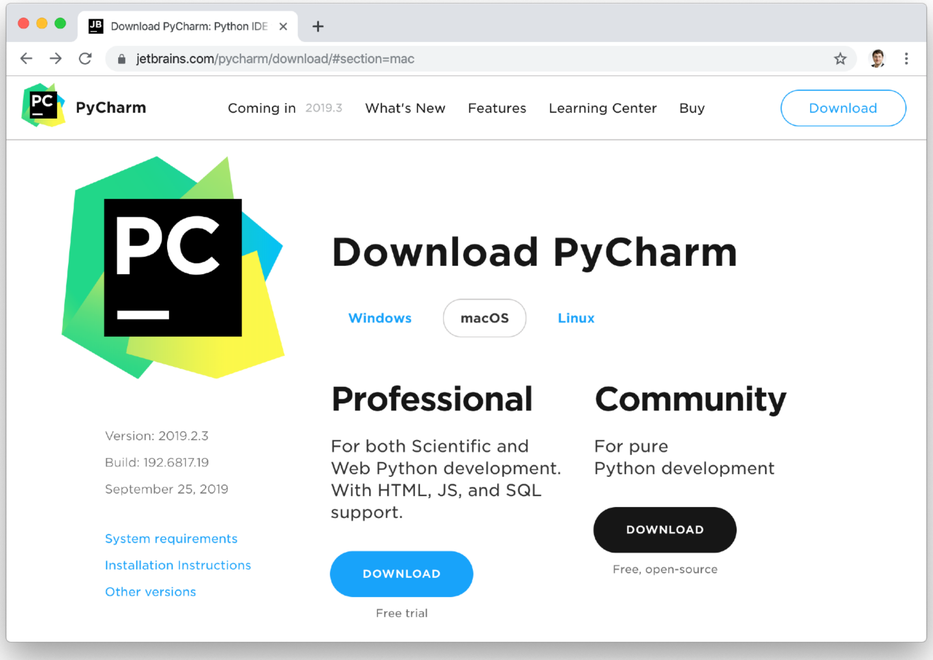
JUPYTER

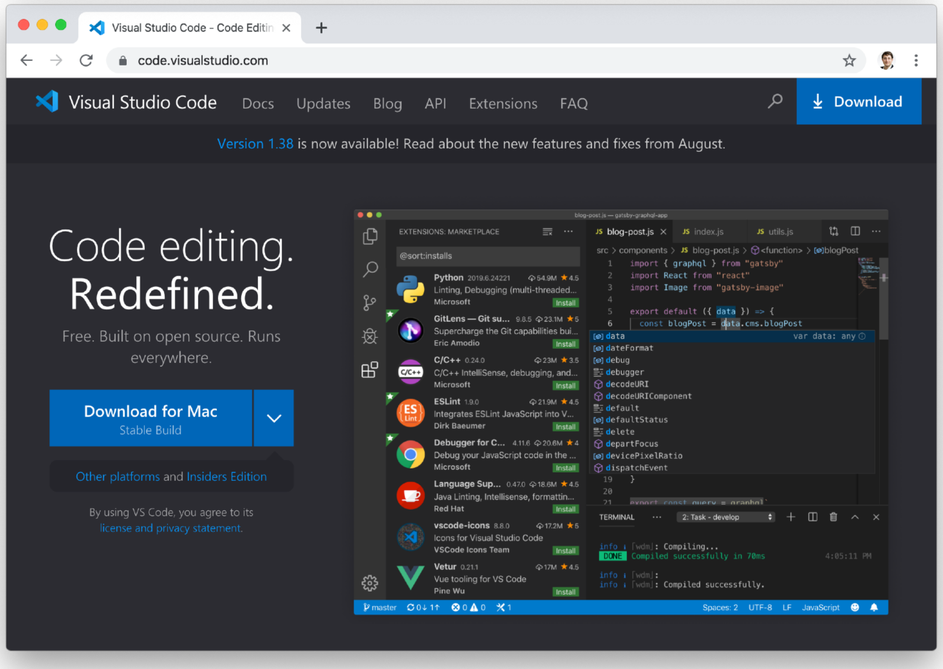
- Jupyter has rich output that's often useful to check data (HTML table, plots)
- But the debugger in Jupyter notebooks is the same as in IPython: ipydb
- A visual debugger?
  - PixieDebugger (from 2018, but I think it doesn't work in JupyterLab)
  - github.com/jupyterlab/debugger
    "A JupyterLab debugger UI extension"
    "In development, not yet available."

IPYTHON & JUPYTER — TIPS

- IPython & Jupyter have ipdb, very similar to PDB (command line interface)
- Generally nicer than Python REPL & PDB — use IPython & Jupyter where available
- Good resources:
  - https://ipython.readthedocs.io
  - https://jupyterlab.readthedocs.io
  - "Errors and debugging" notebook in Data Science Handbook
  - “Wait, IPython can do that?!” by Sebastian Witowski at EuroPython 2019

6. PYCHARM & VS CODE

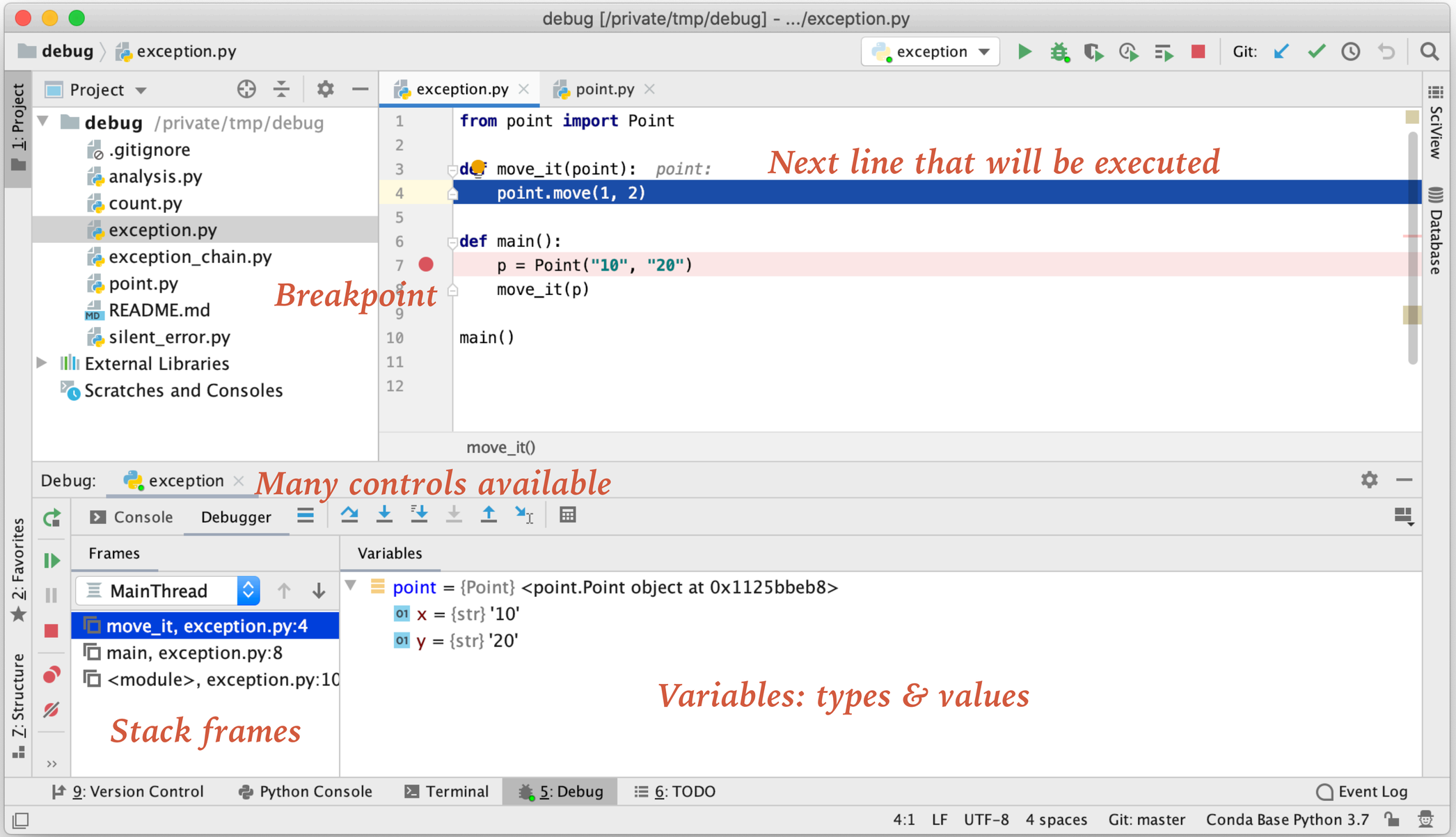




PYCHARM & VS CODE

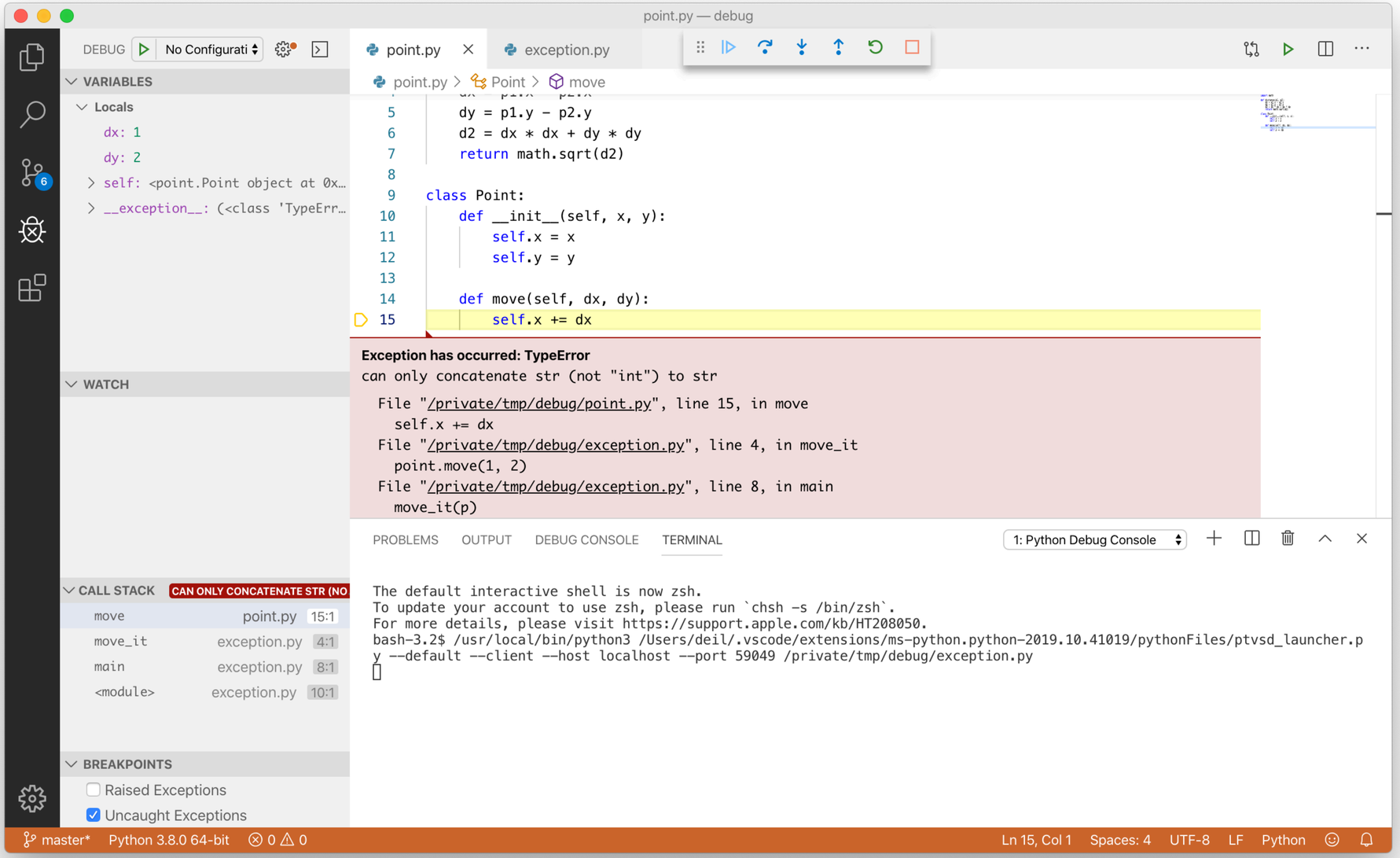
- Many great Python editors & IDEs
- **Key point: visual debugger!**
- I use PyCharm, it's awesome!
  Free community edition has debugger
  Very advanced IDE & code analysis
- VS Code looks great, as well.
  Need to install Python extension extra
  No payed pro version, more lightweight
- *Many others exist: IDLE, emacs, vim, Spyder, Mu, Xcode, Atom, Eclipse, Sublime, …*

PYCHARM VISUAL DEBUGGER — DEMO



VS CODE VISUAL DEBUGGER

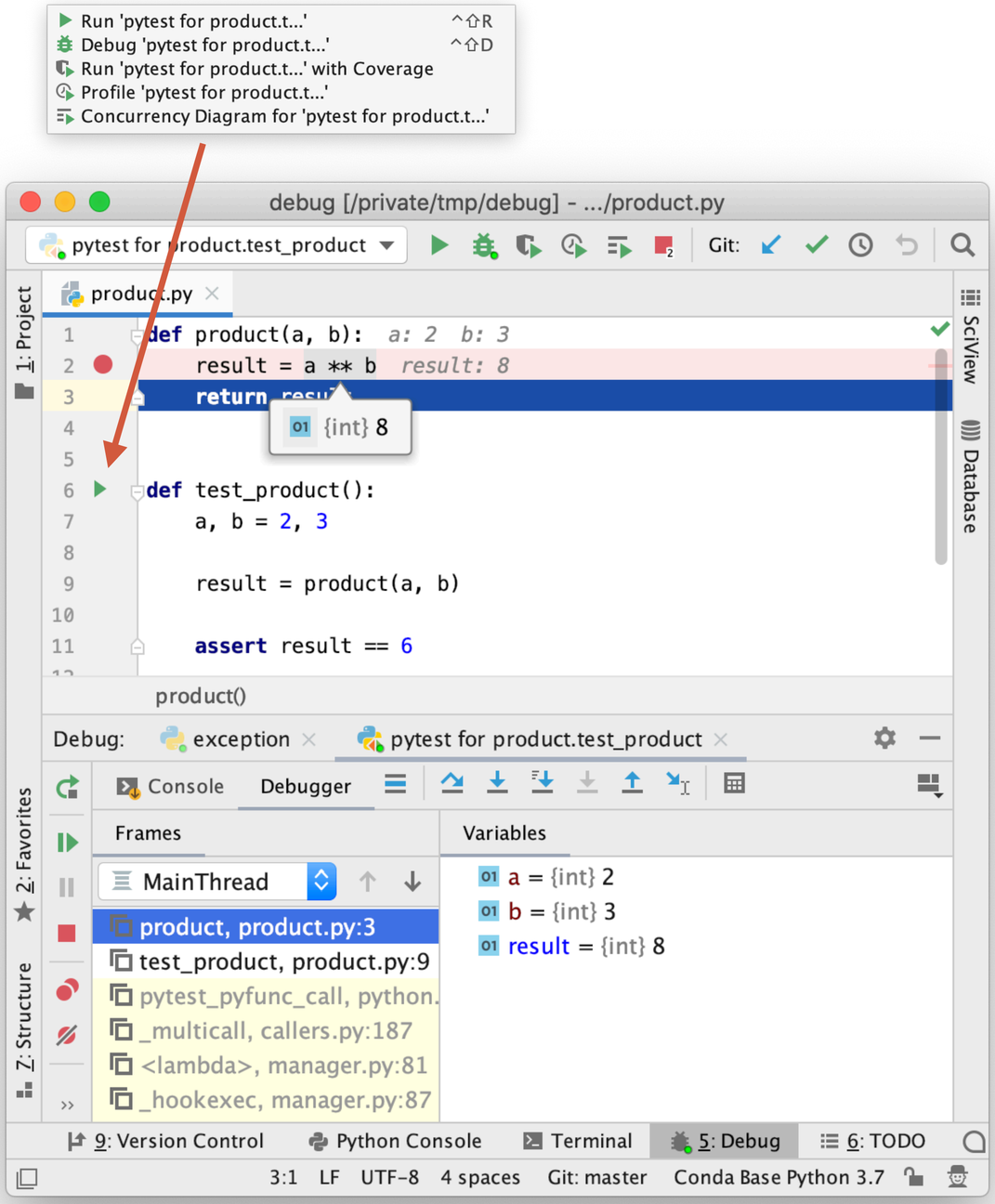
Will not demo. A visual debugger. Very similar to PyCharm.



PYCHARM & VS CODE — TIPS

- Python IDEs offer a **visual debugger**.
- It's awesome, try it! *(Probably easier and more pleasant to learn the PDB)*
- Recommend you try PyCharm and/or VS Code.
- Visit the Jetbrains and Microsoft booth if you have any questions!
- Good resources:
  - Visual debugging in PyCharm by Paul Everitt
    PyCharm Help: Debugging your first Python application
    https://realpython.com/pycharm-guide/
  - https://realpython.com/python-development-visual-studio-code/
    https://code.visualstudio.com/docs/python/python-tutorial

7. TEST



TEST

- Too many bugs & too much debugging?
  - Need systematic effort to improve
  - Add tests: what works and what doesn't?
  - Debug and fix issues via the tests
- Tips:
  - Use **pytest**
  - Use visual test runner & debugger
  - *If you like PDB, use use pytest --pdb*

8. PROFILE

PROFILE

- “Make it run, make it correct, make it fast."
- Use debugging and testing to make it run and make it correct
- If not fast enough or run out of memory:
  - Define a real-world benchmark you care about
  - Measure / “Profile" CPU and RAM usage
  - Try to improve performance (not covered here)
- Let’s look at some profiling tools (there's many more).

```
import psutil
```

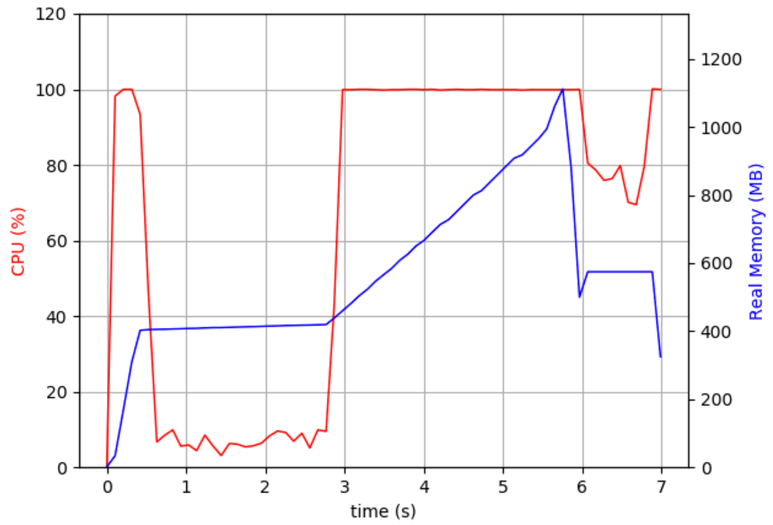
👉 How many CPU cores do you have? What frequency is your CPU?

```
psutil.cpu_count()
psutil.cpu_count(logical=False)
psutil.cpu_freq() # In MHz
```

👉 How much memory do you have? How much free?

```
psutil.virtual_memory().total / 1e9 # In GB
psutil.virtual_memory().free / 1e9 # In GB
```

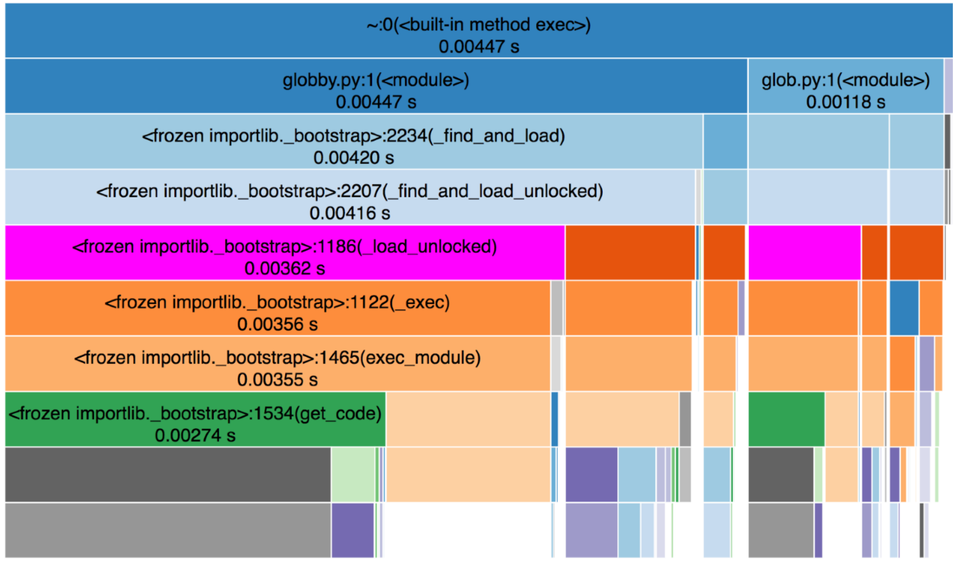
```
$ psrecord --interval 0.1 --plot compute_and_io.png --log compute_and_io.txt 'python compute_and_io.py'
Starting up command 'python compute_and_io.py' and attaching to process
0.000 sec :  starting computation
0.352 sec :  starting network download
2.753 sec :  starting more computation
5.873 sec :  starting disk I/O
6.673 sec :  done
Process finished (7.09 seconds)
```



PSUTIL & PSRECORD

- Process-level profiling:
  - How long does my program take?
  - CPU utilisation (multi-core)?
  - Memory used
- psutil — profile processes (current Python process or any process)
- psrecord — measure CPU and memory usage of a process and make quick plot

```
$ python -m cProfile -o compute.prof compute.py
$ python -m pstats
Welcome to the profile statistics browser.
% help

Documented commands (type help <topic>):
========================================
EOF  add  callees  callers  help  quit  read  reverse  sort  stats  strip

% read compute.prof
compute.prof% stats
Tue Jun  5 17:07:43 2018    compute.prof

         30 function calls in 0.109 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.109    0.109 {built-in method builtins.exec}
       20    0.023    0.001    0.023    0.001 {built-in method builtins.sum}
        2    0.084    0.042    0.084    0.042 compute.py:2(<listcomp>)
        1    0.001    0.001    0.107    0.107 compute.py:10(main)
        1    0.002    0.002    0.109    0.109 compute.py:1(<module>)
        2    0.000    0.000    0.023    0.011 compute.py:4(compute_result)
        2    0.000    0.000    0.084    0.042 compute.py:1(generate_data)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```



CPROFILE & PSTATS & SNAKEVIZ

- Function-level profiling
- Python standard library:
  - cProfile — measure profile
  - pstats — analyse profile
- Snakeviz
  - Third-party tool to visualise and browse profile results (alternative to pstats)

TIMING AND PROFILING FORM IPYTHON & JUPYTER

- `%time` : Time the execution of a single statement
- `%timeit` : Time repeated execution of a single statement for more accuracy
- `%prun` : Run code with the profiler
- `%lprun` : Run code with the line-by-line profiler
- `%memit` : Measure the memory use of a single statement
- `%mprun` : Run code with the line-by-line memory profiler

```
%timeit sum(range(100))
```

```
100000 loops, best of 3: 1.54 µs per loop
```

:

```
%%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
1 loops, best of 3: 407 ms per loop
```

https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html

PROFILING — TIPS

- Process-level profiling: psutil / psrecord
- Function-level profiling: cProfile / pstats / snakeviz
- Line-level profiling: line_profiler
- Profile from IPython & Jupyter: %timeit, %prun, %lprun, %memit, %mprun
- Resources:
  - Timing & Profiling notebook from Python data science handbook
  - Profiling tutorial from me has many examples & links (also debugging tutorial)

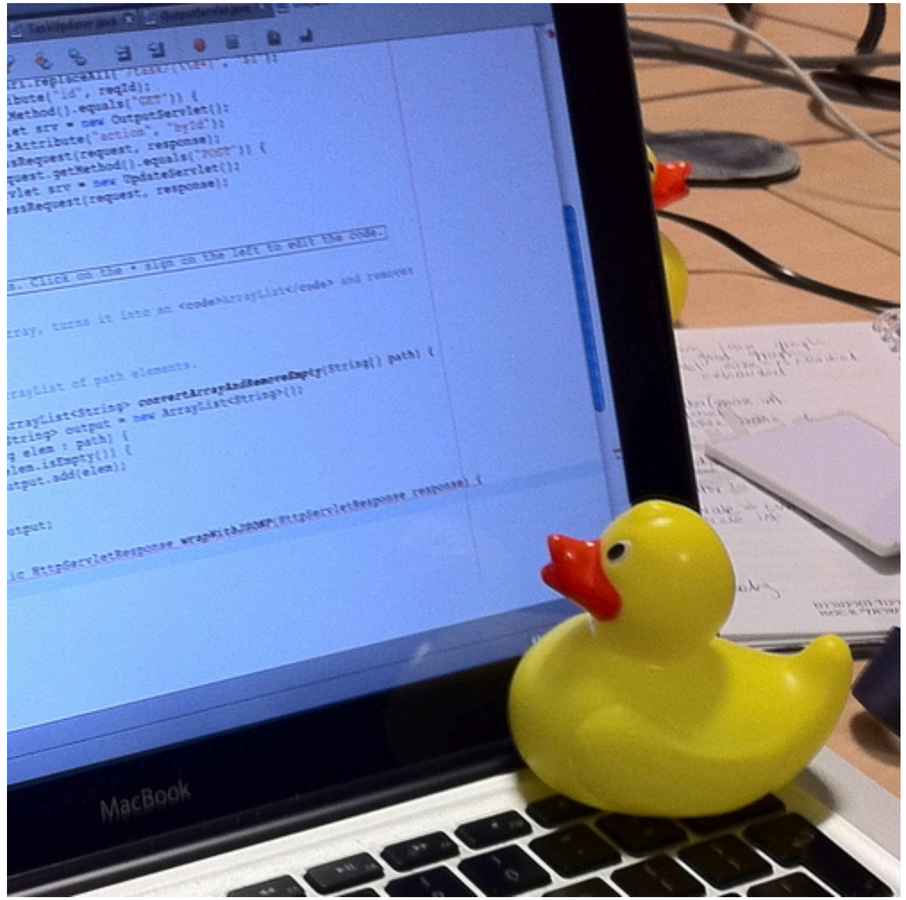
9. LOG

LOGGING

- Logging is useful for long-running programs
- Sometimes the only debug information you can get from production
- Just a quick mention here.
  See https://docs.python.org/3/howto/logging.html

```
import logging
logging.basicConfig(filename='example.log',level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

10. DUCK



- *"Rubber duck debugging" on Wikipedia*

RUBBER DUCK DEBUGGING

- Explain the bug & code to a rubber duck
- If you don't have a duck, use a colleague
- Other general debugging tips:
  - Avoid debugging by writing clean and dumb code and tests.
  - Avoid late-night and long debugging
  - Create a reproducible test case.
    Make it minimal.
    Add as regression test before fixing.

WRAP UP

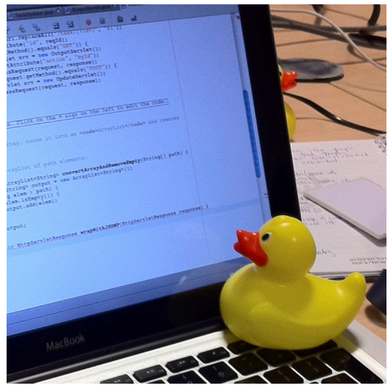
10 WAYS TO DEBUG PYTHON CODE — OVERVIEW

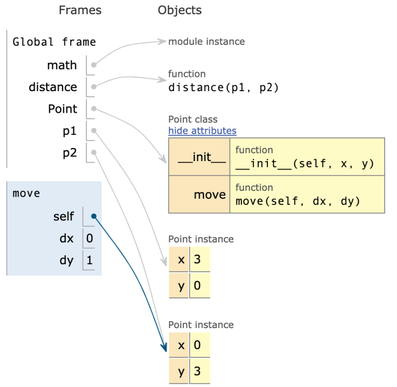
1. Read code
2. Read tracebacks
3. print
4. Python debugger (pdb)
5. IPython & Jupyter
6. PyCharm & VS Code
7. test
8. profile
9. log
10. duck

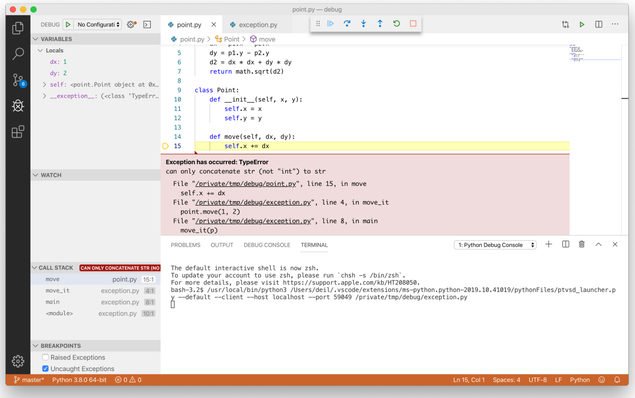
Many topics not covered, e.g. no concurrency, C extensions, web apps.

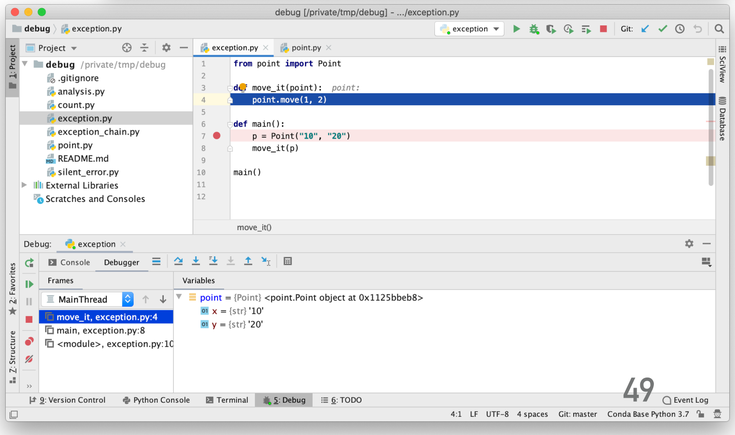
10 WAYS TO DEBUG PYTHON CODE — SUMMARY

- Avoid bugs and debugging as much as possible!
- There will be bugs and debugging!
- Learn to use a debugger!
  - Command line: PDB, IPDB
  - Visual: PyCharm, VS Code

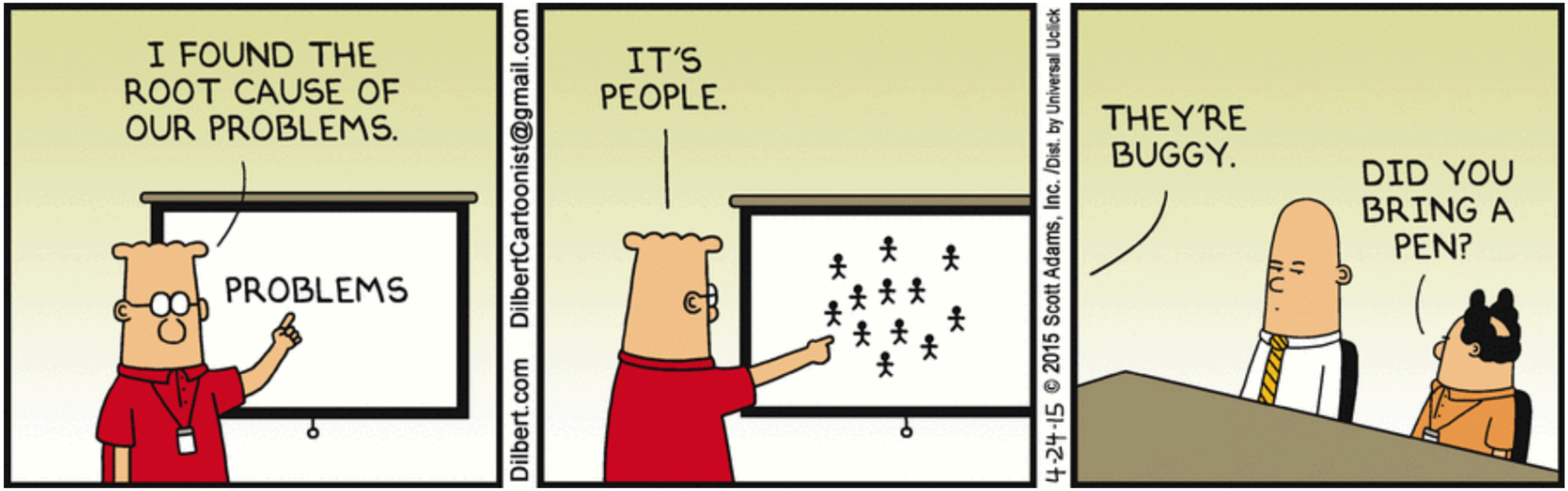








THE END.


I FOUND THE ROOT CAUSE OF OUR PROBLEMS.
PROBLEMS
Dilbert.com DilbertCartoonist@gmail.com
IT'S PEOPLE.
4-24-15 © 2015 Scott Adams, Inc. /Dist. by Universal Uclick
THEY'RE BUGGY.
DID YOU BRING A PEN?