How to Debug Your Python Program

Shengyi Jiang, Qinlin Chen, Yicheng Huang,
Zhiqi Chen & Zhehao Lin

October 18, 2020

Table of Contents

Table of Contents

What can errors tell you

Most Python errors are self-explanatory.

- **SyntaxError**: Improper syntax (e.g. a missing colon or unpaired parentheses/quotes);
- **IndentationError**: Improper indentation (e.g. inconsistent indentation of a function body);
- **TypeError**: Attempted operation on incompatible types (e.g. trying to add a function and a number) or function call with the wrong number of arguments;
- **ZeroDivisionError**: Attempted division by zero;

Cherish Such Kind Error Indications

```
<source>:27:20: error: no member named 'getName' in 'Foo'
                return object.getName();
                       ~~~~~~ ^
<source>:25:8: note: in instantiation of member function 'Object::Model<Foo>::getName' requested here
       Model(const T& t) : object(t) {}
       ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/ext/new_allocator.h:136:23: note: in instantiation of
member function 'Object::Model<Foo>::Model' requested here
       { ::new((void *)__p) _Up(std::forward<_Args>(__args)...); }
                       ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/alloc_traits.h:475:8: note: in instantiation of
function template specialization '__gnu_cxx::new_allocator<Object::Model<Foo> >::construct<Object::Model<Foo>, const Foo>' requested here
       { __a.construct(__p, std::forward<_Args>(__args)...); }
             ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr_base.h:526:30: note: in instantiation of
function template specialization 'std::allocator_traits<std::allocator<Object::Model<Foo> > >::construct<Object::Model<Foo>, const Foo>' requested here
         allocator_traits<_Alloc>::construct(__a, _M_ptr(),
                                  ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr_base.h:637:18: note: in instantiation of
function template specialization 'std::_Sp_counted_ptr_inplace<Object::Model<Foo>, std::allocator<Object::Model<Foo> >,
__gnu_cxx::_S_atomic>::_Sp_counted_ptr_inplace<const Foo>' requested here
          ::new (__mem) _Sp_cp_type(std::move(__a),
                 ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr_base.h:1294:14: note: (skipping 1 context
in backtrace; use -ftemplate-backtrace-limit=0 to see all)
       : _M_ptr(), _M_refcount(__tag, (_Tp*)0, __a,
                  ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr.h:344:4: note: in instantiation of
function template specialization 'std::__shared_ptr<Object::Model<Foo>, __gnu_cxx::_S_atomic>::__shared_ptr<std::allocator<Object::Model<Foo> >, const
Foo>' requested here
       : __shared_ptr<_Tp>(__tag, __a, std::forward<_Args>(__args)...)
         ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr.h:690:14: note: in instantiation of
function template specialization 'std::shared_ptr<Object::Model<Foo> >::shared_ptr<std::allocator<Object::Model<Foo> >, const Foo>' requested here
      return shared_ptr<_Tp>(_Sp_make_shared_tag(), __a,
                 ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr.h:706:19: note: in instantiation of
function template specialization 'std::allocate_shared<Object::Model<Foo>, std::allocator<Object::Model<Foo> >, const Foo>' requested here
```

Figure: A Typical C++ Template Error

Do Not Be Afraid of Seeing Errors

```
def a_plus_abs_b(a, b):
    if b >= 0:
        h = a + b
    else:
        h = a - b
    return h(a, b)
```

```
  File "*", line 23, in <module>
    a_plus_abs_b(2, 3)
  File "*", line 21, in a_plus_abs_b
    return h(a, b)
TypeError: 'int' object is not callable
```

The error message indicates the location (File and Line No.) and type (TypeError) of your error. The text `'int' object is not callable` also tells you that `h` should be a function.

Note: You can Baidu or Google the error if you do not understand its meaning.

Errors are not enough

Sometimes, the program crashes long after your actual error (or does not crash but gives you a wrong answer). In such cases, errors may not provide useful information and sometimes lead you to a wrong direction.

Interesting bugs in real life.

Table of Contents

Debug using debugger

Debugger

A debugger or debugging tool is a computer program used to test and debug other programs (the "target" program). Typical debugging facilities include the ability to

- run or halt the target program at specific points
- display frame contents
- modify frame contents

Notes: We will show examples about using debugger in PyCharm(docs). Things are similar in VSCode, so you may find the usage yourself. You can also try to use debugger in terminal (docs for `pdb`).

Add a configuration in PyCharm

PyCharm does not support debugging a `doctest` directly (You can try to debug a doctest and see what will happen). We need to add a new Python configuration for debugging. The general process is similar to adding a `doctest` configuration.

Add a configuration in PyCharm

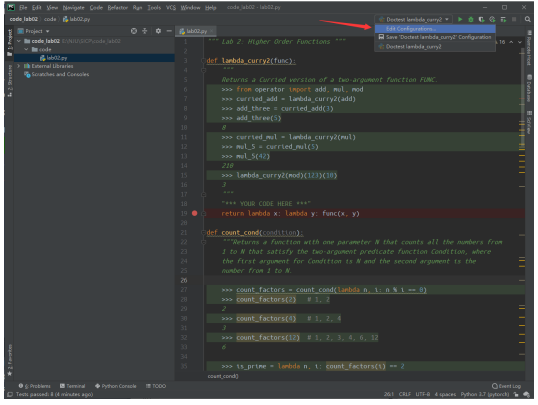


Figure: Click `Edit Configurations`

Add a configuration in PyCharm

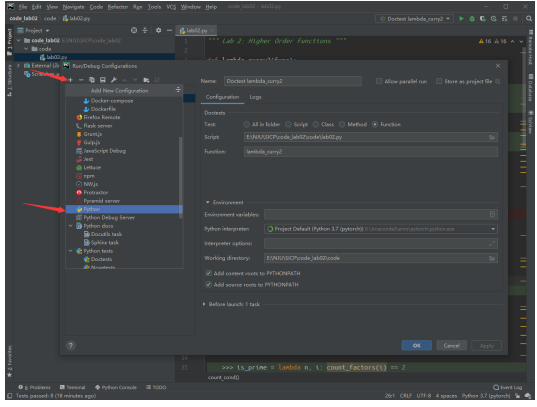


Figure: Click + and choose `Python`

Add a configuration in PyCharm

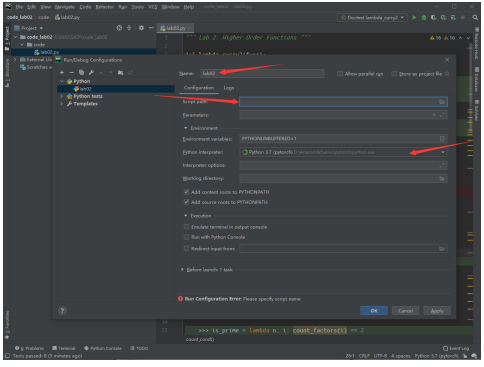


Figure: Change `Name`, `Scripts path`, `Python interpreter`.

You can use any `Name`, but `Scripts path`, `Python interpreter` must be correct.

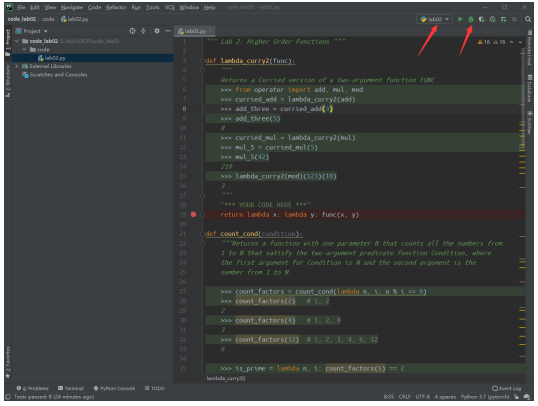


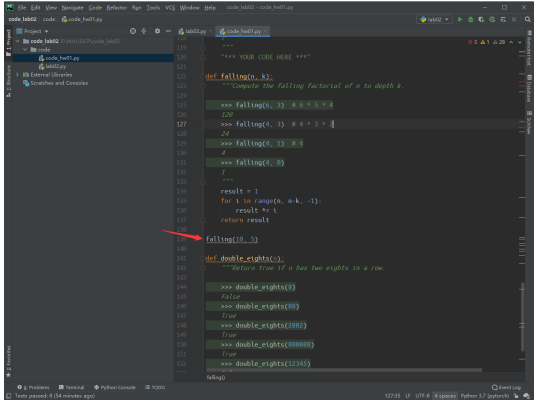
Figure: Select the configuration you just created and click `Debug` button

Hmm, nothings happens. Why?
You do not actually CALL any functions in the script.

Note: `doctest` does the function call automatically to check if your function output matches the expected one.

Make a Function Call

Explicitly add a function call to the function you have implemented. You can choose `doctest` examples or design some new cases as function inputs.



Add a Breakpoint

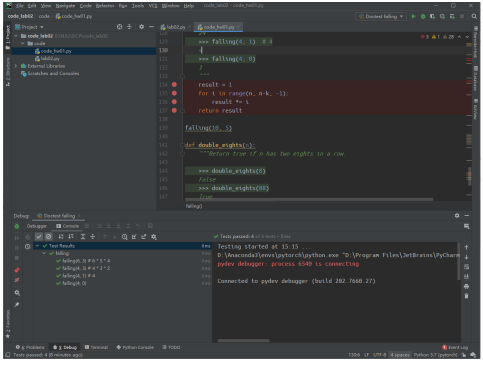


Figure: Click the space right to the line no. to add a breakpoint. Click the circle to cancel it.

Breakpoint: debugger will halt the program when reaching a breakpoint.

Control the debugging process

- Resume: Resume a debug session (stop at the next breakpoint).
- Pause: Pause the code and show the current execution point.
- Step Over: Steps over the current line of code and takes you to the next line even if the highlighted line has method calls in it.
- Step Into: Steps into the method to show what happens inside it.
- Step Into My Code: Same as above, but ignores code of thirdparty libraries.
- Force Step Into: Steps in the method even if this method is skipped by the regular Step Into.
- Step Out: Steps out of the current method and takes you to the caller method.
- Run To Cursor: Continues the execution until the position of the caret is reached.

Make Debugging Great Again

Select the configuration you just created and click `Debug` button AGAIN.

What can we get from debugging?

Variable Value

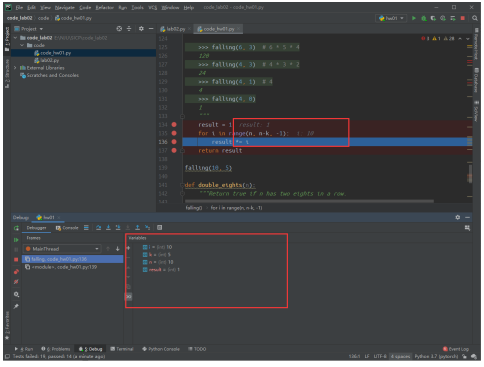


Figure: PyCharm will display values of variables in selected regions.

Call Stack

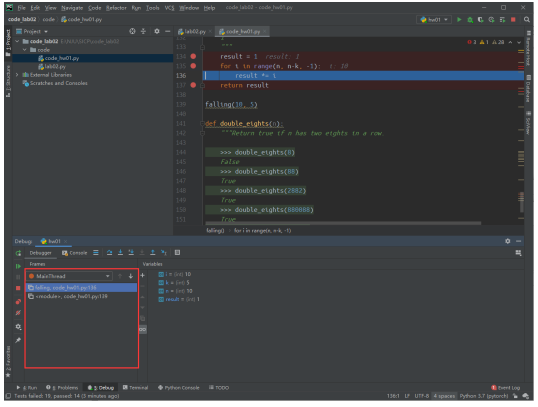


Figure: PyCharm will display call stack in selected regions.

Evaluate Expression

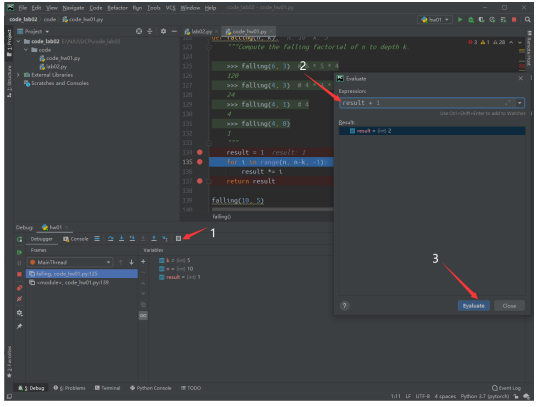


Figure: Evaluate an arbitrary expression.

Run statements

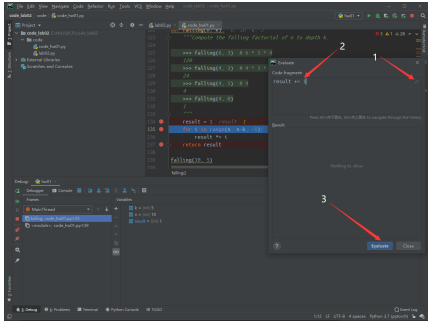


Figure: Run arbitrary code.

Note: Statements may change your program frames. For example, if you evaluate `result += 1`, `result` in your program is bound to 2

Evaluate Expressions and Run Code in Console

You can also switch to console panel to execute expressions and run code.

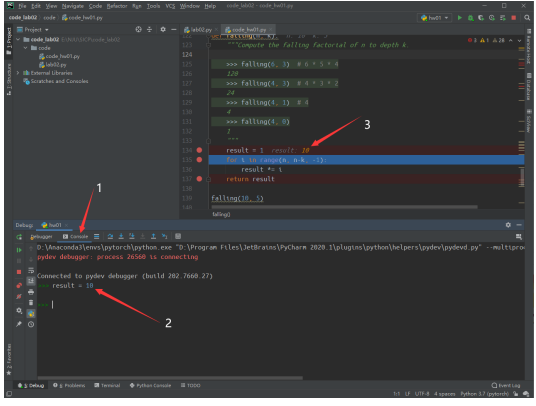


Table of Contents

Debug using print/assert

Debugger is handy. Why bother?

- Debugging step by step is too slow;
- Programs in debugging mode runs slowly;
- Your cannot use a debugger
  - Your program is concurrent
  - Your program runs in a system where no debugger is not supported
- You can focus on a specific variable
- You do not know how to use a debugger (go back to the Sec 2 and other materials :)

Main idea:
Insert `print/assert` in your code to see if intermediate results match your expectation.