Debugging

“Beware of bugs in the above code; I have only proved it correct, not tried it.”
-David Knuth

assert

Assertions: Use

- What happens if you run `half_fact(5)`?
    - Infinite loop??????
- Code should fail as soon as possible
    - Makes error detection easier
- Assertions are forever

```python
def fact(x):
    assert isinstance(x, int)
    assert x >= 0
    if x == 0:
        return 1
    else:
        return x * fact(x - 1)


def half_fact(x):
    return fact(x / 2)
```

Assertions: Limitations

- Require invariants
  - Assertions tend to be useful when you know a good invariant
  - An **invariant** is something that is always true
  - E.g., the argument to fact being a non-negative integer
- Assertions *check* that code meets an *existing* understanding
  - They are less useful at actually developing an understanding of how some code is working
  - Generally, assertions are best added to your *own* code, not someone else's
  - (For the purpose of debugging, you six months ago is a different person)

Assertions: Limitations demo

- What assertion should be added here?

```
def t(f, n, x, x0=0):
    assert ????
    r = 0
    while n:
        r += (x-x0) ** n / fact(n) * d(n, f)(x0)
        n -= 1
    return r
```

Testing

Testing: Why do it?

- *Detect errors* in your code
- *Have confidence* in the correctness of subcomponents
- *Narrow down* the scope of debugging
- *Document* how your code works

Testing: Doctests

- Python provides a way to write tests as part of the docstring
- Just put the arrows and go!
- Right there with the code and docs

- To run:
    - python3 -m doctest file.py

```
# in file.py
def fib(n):
    """Fibonacci

    >>> fib(2)
    1
    >>> fib(10)
    55
    """
     ...
```

Testing: Doctest Limitations

- Doctests have to be in the file
  - Can’t be too many
- Do not treat print/return differently
  - Makes print debugging difficult
  - ok fixes this issue

```
def fib(n):
    """Fibonacci

    >>> fib(2)
    1
    >>> fib(10)
    55
    >>> fib(0)
    0
    >>> fib(3)
    2
    >>> fib(4)
    3
    >>> fib(8)
    21
    >>> fib(5)
```

Print Debugging

Print Debugging: Why do it?

- Simple and easy!
- Quickly gives you an insight into what is going on
- Does not require you to have an invariant in mind

```
def fact(x):
    ~~assert isinstance(x, int)~~
    ~~assert x >= 0~~
    print("x =", x)
    if x == 0:
        return 1
    else:
        return x * fact(x - 1)

def half_fact(x):
    return fact(x / 2)
```

Print Debugging: ok integration

- The code on the right doesn't work, if you have an ok test for `fact(2)`

```
Error: expected
    2
but got

    x= 2
    x= 1
    x= 0
    2
```

```
def fact(x):
    ~~print([illegible]” [illegible]x)~~
    print(“Debug: x=”, x)
    if x == 0:
        return 1
    else:
        return x * fact(x - 1)

def half_fact(x):
    return fact(x / 2)
```

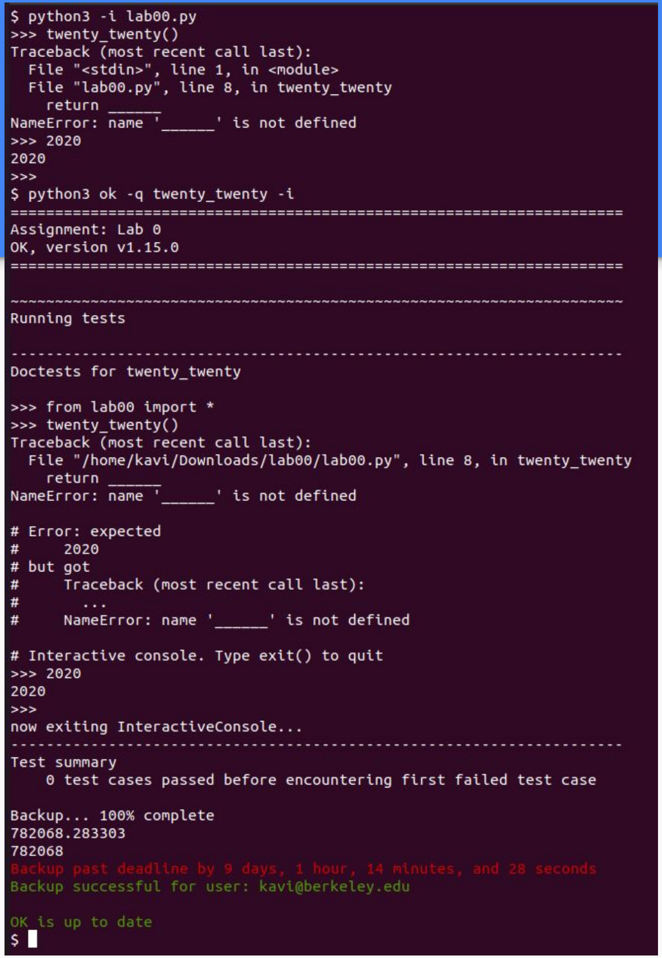
Interactive Debugging

Interactive Debugging

- Sometimes you don’t want to run the code every time you change what you choose to print
- Interactive debugging is live

Interactive Debugging: REPL

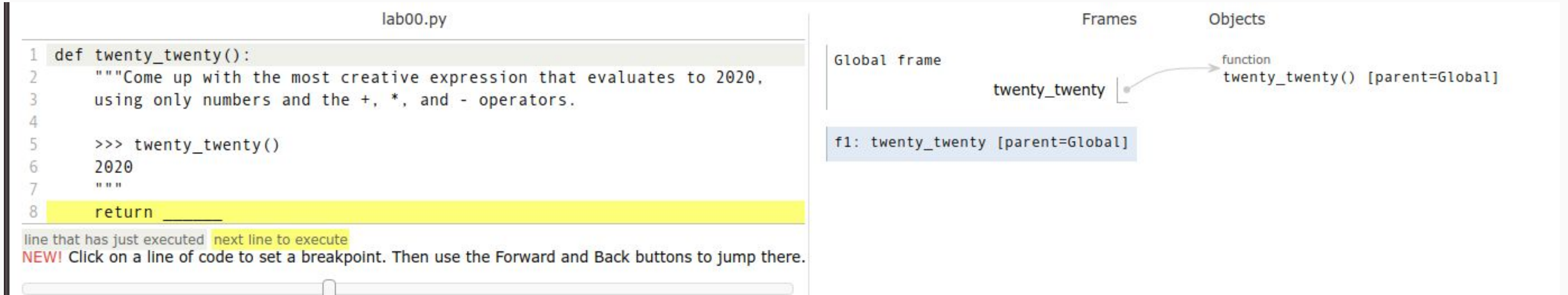
- The interactive mode of python, known as the REPL, is a useful tool
- To use, run
  - `python3 -i file.py`
  - then run whatever python commands you want
- OK integration:
  - `python3 ok -q whatever -i`
  - Starts out already having run code for that question



```
$ python3 -i lab00.py
>>> twenty_twenty()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lab00.py", line 8, in twenty_twenty
    return ______
NameError: name '______' is not defined
>>> 2020
2020
>>>
$ python3 ok -q twenty_twenty -i
=====================================================================
Assignment: Lab 0
OK, version v1.15.0
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Running tests

---------------------------------------------------------------------
Doctests for twenty_twenty

>>> from lab00 import *
>>> twenty_twenty()
Traceback (most recent call last):
  File "/home/kavi/Downloads/lab00/lab00.py", line 8, in twenty_twenty
    return ______
NameError: name '______' is not defined

# Error: expected
#     2020
# but got
#     Traceback (most recent call last):
#       ...
#     NameError: name '______' is not defined

# Interactive console. Type exit() to quit
>>> 2020
2020
>>>
now exiting InteractiveConsole...
---------------------------------------------------------------------
Test summary
    0 test cases passed before encountering first failed test case

Backup... 100% complete
782068.283303
782068
Backup past deadline by 9 days, 1 hour, 14 minutes, and 28 seconds
Backup successful for user: kavi@berkeley.edu

OK is up to date
$
```

Interactive Debugging: PythonTutor

- You can also step through your code line by line on PythonTutor
  - Just copy your code into tutor.cs61a.org
- Ok integration
  - python ok -q whatever --trace



lab00.py

```
def twenty_twenty():
    """Come up with the most creative expression that evaluates to 2020,
    using only numbers and the +, *, and - operators.

    >>> twenty_twenty()
    2020
    """
    return ______
```

line that has just executed next line to execute

NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there.

Frames
Objects
Global frame
twenty_twenty
function
twenty_twenty() [parent=Global]
f1: twenty_twenty [parent=Global]

Error Types

Error Message Patterns

- Ideally: this wouldn’t be necessary
  - Error messages would clearly say what they mean
- In practice, error messages are messy
- Not universal laws of nature (or even Python)
  - Good guidelines that are true >90% of the time

SyntaxError

- What it technically means
  - The file you ran isn’t valid python syntax
- What it practically means
  - You made a typo
- What you should look for
  - Extra or missing parentheses
  - Missing colon at the end of an if or while statement
  - You started writing a statement but forgot to put anything inside

IndentationError

- What it technically means
  - The file you ran isn't valid python syntax, because of indentation inconsistency
- What it practically means
  - You used the wrong text editor
- What you should look for
  - You made a typo and misaligned something
  - You accidentally mixed tabs and spaces
    - Open your file in an editor that shows them
  - You used the wrong kind of spaces
    - Yes, there is more than one kind of space
    - If you think this is what's going on, post on piazza with a link to the okpy backup

TypeError: … ‘X’ object is not callable ...

- What it technically means
  - Objects of type X cannot be treated as functions
- What it practically means
  - You accidentally called a non-function as if it were a function
- What you should look for
  - Variables that should be functions being assigned to non-functions
  - Local variables that do not contain functions having the same name as functions in the global frame

TypeError: … NoneType ...

- What it technically means
  - You used None in some operation it wasn’t meant for
- What it practically means
  - You forgot a return statement in a function
- What you should look for
  - Functions missing return statements

NameError or UnboundLocalError

- What it technically means
  - Python looked up a name but didn't find it
- What it practically means
  - You made a typo
- What you should look for
  - A typo in the name in the description
  - *(less common)* Maybe you shadowed a variable from the global frame in a local frame (see right)

```
def f(x):
    return x ** 2

def g(x):
    y = f(x)
    def f():
        return y + x
    return f
```

Tracebacks

Parts of a Traceback

```
def f(x):
    1 / 0
def g(x):
    f(x)
def h(x):
    g(x)
print(h(2))
```

- Components
  - The error message itself
  - Lines #s on the way to the error
  - What's on those lines
- Most recent call is at the bottom

```
Traceback (most recent call last):
  File "temp.py", line 7, in <module>
    print(h(2))
  File "temp.py", line 6, in h
    g(x)
  File "temp.py", line 4, in g
    f(x)
  File "temp.py", line 2, in f
    1 / 0
ZeroDivisionError: division by zero
```

How to read a traceback

```
def f(x):
    1 / 0
def g(x):
    f(x)
def h(x):
    g(x)
print(h(2))
```

1. Read the error message
   a. Remember what common error messages mean!
2. Look at each line, bottom to top and see which one might be causing it

```
Traceback (most recent call last):
  File "temp.py", line 7, in <module>
    print(h(2))
  File "temp.py", line 6, in h
    g(x)
  File "temp.py", line 4, in g
    f(x)
  File "temp.py", line 2, in f
    1 / 0
ZeroDivisionError: division by zero
```