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 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 in 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?

```python
def t(f, n, x=0):
    assert ???
    r = 0
    while n:
        r += (x=x) ** n / fact(n) * d(n, f(x))
        n -= 1
    return r
```
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 documentation
- Just put the arrows and go!
- Right there with the code and docs
- To run:
  - `python3 -m doctest file.py`

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
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

# in file.py
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

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):
    if x <= 1:
        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)`

  ```python
  Error: expected 2
  but got
  x= 2
  x= 1
  x= 0
  2
  ```

  ```python
  Error: expected 2
  but got
  x= 2
  x= 1
  x= 0
  2
  ```
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 whatever -t
  - Starts out already having run code for that question

Interactive Debugging: PythonTutor

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

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

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

How to read a traceback

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
   a. Look for the line containing the error message

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