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
  ```python
def half_fact(x):
  assert x >= 0
  if x == 0:
    return 1
  else:
    return x * fact(x - 1)
  ```
- Code should fail as soon as possible
  - Makes error detection easier
- Assertions are forever

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:
  - `python-rn` 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 to fix this issue

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 if 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 fib(n, x, x0=0):
    assert x0 == 0
    while n:
        x *= (x-x0)**n / fact(n) * (2**(n-1))
        x -= x
    return x
```
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

```python
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)
- 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`
- Run whatever python commands you want
- Ok integration:
  - `python3 ok -q whatever -i`
  - 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 [PythonTutor](https://pythontutor.com)
- Ok integration:
  - `python -m ok` whatever `-i`

Error Types

- Ideally: this wouldn't be necessary
- In practice, error messages are messy
- Not universal laws of nature (or even Python)
  - Good guidelines that are true >90% of the time

Error Message Patterns
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 oily 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)

Tracebacks
- Components
  - The error message itself
  - Lines #3 on the way to the error
  - What’s on those lines
  - Most recent call is at the bottom

Parts of a Traceback
- Traceback (most recent call last):
  - File 'temp.py', line 7, in <module>
  - File 'temp.py', line 6, in h
  - File 'temp.py', line 5, in g
  - File 'temp.py', line 3, in f
  - 5 / 0
  - ZeroDivisionError: division by zero

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