



# SSW-810: Software Engineering Tools and Techniques

## *Python Fundamentals*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

And

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

# Today's topics

## Python Overview

Variables, statements, expression

Operators/Operands

Functions

*Booleans, conditionals*

*If/elif/else*

*While/for*

*List, dict containers*



## Integrated Development Environments (IDE)



# Just for fun...

## CHOOSE YOUR WEAPON...



C++



JAVA



PYTHON



C  
bladecon



# Talking to Python

Say hello!

```
print('hello world')
```

You type the stuff in the  
pink/grey box

hello world

Output from Python

Python is an interpreted language

Commands are parsed and executed interactively

No need to compile

If you're unsure of what Python would do, try it!!!

Designing Python experiments is a critical skill



# Comments: Note to self!

Comments allow us to leave reminders for us and others who read our code

```
# this is a comment  
# Python ignores comments  
print('Hello world!')
```

Hello world!

```
""" This is a docstring """  
''' this is also a docstring '''
```

Comments are **critical** for readable, maintainable code

Code lives far longer than we expect, and comments help us to understand what we were thinking after the fact



# Python primitives (Py ingredients)

2 An integer number

3.14157 A floating point number

'Hello world!'

Two equivalent ways to specify a string

"Hello world!"

"I'm a string with an apostrophe"

Double quotes make it  
easy to include  
apostrophes

'A string with "double" quotes'

""" Docstrings are strings, but also statements """



# Dynamic typing

Values have **types** that are determined dynamically

You can ask Python to identify the type of a value

```
type(2)
```

int

```
type(3.14157)
```

float

```
type('hello world')
```

str

```
type('2')
```

What? Why not an int???

str



# Optional Type Hints

Python 3.6+ allows optional type hints to specify the expected type of variables and functions

```
an_integer: int = 2  
a_float: float = 3.14  
a_str: str = "hello world"
```

Plus1 expects to be passed one int and returns an int

```
def plus1(n: int) -> int:  
    return n + 1
```

```
an_integer = "goodbye"
```

What??? We defined 'an\_integer' to be an int

Python **ignores** the type hints at run time

IDEs use the type hints to warn about potential problems



# Optional Type Hints

Type hints specify the data type of variables, parameters and the value returned by functions

If you need ...	Example	Use type:
string	"hello world"	str
integer	42	int
float	3.14	float
integer or float	42 or 3.14	float
Boolean	True, False	bool
Function returns None or no value		None

We'll look at many other types,  
including user defined types



# Why specify type hints?

Type hints help to document the code and automatically detect problems

Users > jrr > Downloads > why\_type\_annotations.py > ...

```
1 def factorial(n: int) -> int:
2     """ return n! """
3     print(f"calculating factorial({n})")
4     if n == 1:
5         return 1
6     else:
7         return n * factorial(n - 1)
8
9 print(f"3! = {factorial(3)}") # valid call to compute 3!
10 print(f"'3'! = {factorial('3')}") # invalid call: should pass int, but passed str
11 print(f"3.1! = {factorial(3.1)}") # factorial(float) may cause stack overflow if n never == 1
12
```

Factorial() expects one int  
and returns an int

PROBLEMS 2 OUTPUT DEBUG CONSOLE ... Filter: E.g.: text, \*\*/\*.ts, !\*\*/node\_modules/\*\*

✓ why\_type\_annotations.py ~/Downloads 2

- ✗ Argument 1 to "factorial" has incompatible type "str"; expected "int" mypy(error) [10, 28]
- ✗ Argument 1 to "factorial" has incompatible type "float"; expected "int" mypy(error) [11, 28]

Identified two problems without running the code!



# Python keywords (reserved words)

Python reserves some words that can't be used as names

and	as	assert	break	class
continue	def	del	elif	else
except	exec	finally	for	from
global	if	import	in	is
lambda	nonlocal	not	or	pass
raise	return	try	while	with
yield	True	False	None	



# Python statements

Statements allow us to tell Python what we want to do

## Examples of statements

```
cnt: int = 42

while cnt < 50:
    cnt += 1

if cnt == 51:
    print(51)

for item in ['donut', 'bagel']:
    print("I'm hungry for a", item)
```

```
I'm hungry for a donut
I'm hungry for a bagel
```

The Python interpreter (REPL) evaluates statements

Statements don't produce results

Python interpreter doesn't print anything after statements



# Python expressions

Expressions are combinations of values, variables, and function calls

```
cnt: int = 42
```

This is a statement so no output

```
cnt
```

Evaluate the expression cnt

```
42
```

```
len("Hello world")
```

Call the len function

```
11
```

```
3 * 4 + 5
```

Evaluate the expression

```
17
```



# Assignment statements

```
n: int = 42  
n = n + 1
```

Python provides syntactic short cuts to improve readability

Statement	Equivalent statement
<code>n = n + 1</code>	<code>n += 1</code>
<code>n = n - 1</code>	<code>n -= 1</code>
<code>n = n / 2</code>	<code>n /= 2</code>
<code>n = n * 2</code>	<code>n *= 2</code>
<code>n = n % 2</code>	<code>n %= 2</code>
<code>n = n // 10</code>	<code>n //= 10</code>

Wait! What's that?  
How can we discover  
the semantics of '%'?

Python does **not** support `n++`, `++n`, `n--`, `--n`



# Getting input from the user

Python provides a function to get input from a user

```
response: str = input("Please enter a number:")
```

```
Please enter a number:42
```

```
response
```

```
'42'
```

Input always returns a string

```
response + 1
```

```
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-27-d35ac03d859c> in <module>
----> 1 response + 1
```

Python is unhappy ;-)

```
TypeError: can only concatenate str (not "int") to str
```

```
int(response) + 1
```

Cast strings to integers or floats to do math



# Printing output to the user

Python's print function writes output to the user

```
name: str = "Jim"  
number: int = 42  
print(f"My name is {name} and my favorite number is {number}")
```

f"..." {expr} ..." replaces {expr} with the value  
of {expr} in the formatted string

My name is Jim and my favorite number is 42

```
print("My name is", name, "and my favorite number is", number)
```

My name is Jim and my favorite number is 42

Print accepts any number of arguments of many types

Don't forget the 'f'  
f"..." {expr} ..."



# Functions

## Functions are a sequence of statements

```
def name(parameters) -> return_type:  
    """ parameters has parameter_1: type_1, parameter_2: type_2, ... """  
    statements
```

Where:

**name**: the function name which must be a legal identifier

**parameters**: 0 or more identifiers separated by commas.

These parameters are available as variables inside the function

**return\_type**: the type returned by the function

**statements**: one or more statements

Functions may include an optional [return statement](#)

Functions **always** return a value (**None** by default)

# Functions: *hello*

“Parameters” in the function definition can be used in the function definition

```
def hello(name: str) -> str:  
    return f"Hello {name}!"
```

```
hello("Nanda")  
'Hello Nanda!'
```

Name has value “Nanda” inside function hello()

Values passed into functions are called “arguments”



# Functions: $n^2$

“Parameters” in the function definition

```
def square(n: int) -> int:  
    return n * n
```

The parameters are available as variables inside the function

```
square(2)
```

n has value 2

4



# Functions: $n^2$

What if we want calculate square(int) and square(float)?

```
def square(n: float) -> float:  
    return n * n
```

```
square(3.14)
```

```
9.8596
```

float type covers both  
float and int

```
square(2)
```

```
4
```

One function (and type hint) handles both float and int



# Function definitions

Python uses whitespace to define blocks

Use whitespace consistently

4 spaces is standard to optimize readability

Most IDEs automatically convert tab to 4 spaces

```
def name(parameters) -> return_type:  
    statement_1  
    statement_2  
    statement_n
```

4 spaces



# Python whitespace

You must consistently indent or Python won't understand your function definition

Some languages use { ... } e.g. C, C++, Java

Python uses indenting to define related blocks of code

Use 4 spaces to indent

```
def factorial(n: int) -> int:  
    """ return n! == n*n-1*n-2*... """  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

No semi-colons or  
curly braces in Python

**Indent carefully to avoid frustration!**



# Documenting functions

Every file and function should begin with a **docstring**

```
def factorial(n: int) -> int:  
    """ return n! == n*n-1*n-2*... """  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

docstring

```
help(factorial)
```

A function's docstring is available  
from the help() function

Help on function factorial in module `__main__`:

```
factorial(n: int) -> int  
    return n! == n*n-1*n-2*...
```



# Fruitful functions

**Every function returns a value:**

`return` statement returns an explicit value

Falling out the bottom of the function returns `None`

Not all functions calculate a value to return to the caller

`print('Hello world')`

Print returns `None` but we don't use the return value or care

Functions that return a useful value are called ***fruitful*** functions

Some functions exist for their side effects, e.g. printing



# Your turn

What does this function return?

```
def foo(n: int) -> int:  
    print('Called foo')  
    n + 1  
  
result = foo(3)  
result = ???
```

What is result?

3? No

4? No

None? Yes

No explicit return returns None



# Recursive Functions

A python functions can call other functions  
Including itself (recursion)

$$4! = 4 * 3 * 2 * 1$$

$$3! = 3 * 2 * 1$$

$$2! = 2 * 1$$

$$1! = 1$$

```
def factorial(n: int) -> int:  
    """ return n! == n*n-1*n-2*... """  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n - 1)
```





# Boolean values

Python includes a Boolean type (`bool`)

A Boolean value is either `True` or `False`

```
5 < 8
```

`True`

```
5 > 8
```

`False`

```
5 == (3 + 2)
```

`True`



# Functions can return True/False

It's sometimes useful to define functions that return True/False for use as Boolean operands

```
def is_even(n: int) -> bool:  
    return n % 2 == 0  
  
def is_odd(n: int) -> bool:  
    return n % 2 == 1
```

```
is_even(42)
```

True

```
is_odd(42)
```

False

# Comparison operators

Conditionals help to make decisions

Comparison Operator	Semantics	Example
<code>==</code>	Equal	<code>3 == 4</code>
<code>!=</code>	Not equal	<code>3 != 4</code>
<code>&lt;</code>	Less than	<code>'a' &lt; 'b'</code>
<code>&gt;</code>	Greater than	<code>'a' &gt; 'b'</code>
<code>&lt;=</code>	Less than or equal to	<code>3 &lt;= 4</code>
<code>&gt;=</code>	Greater than or equal to	<code>3 &gt;= 4</code>



# Logical operators

Combine logical expressions with logical operators:

and, or, not

```
5 < 8 and 10 > 12 # both must be True
```

False

```
5 > 8 or 7 < 9 # only one must be True
```

True

```
not 3 < 2
```

True



# Truth tables for logical operators

x	y	x and y <sup>1</sup>	x or y <sup>2</sup>
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

<sup>1</sup> Both must be True or x and y is False

<sup>2</sup> Only one of x, y must be True for x or y to be True



# Short circuiting

Python evaluates logical expressions from left to right

If the **left** operand of an `or` operator is True then the right operand is **not** evaluated and the expression is True

If the **left** operand of an `or` operator is False then both operands are evaluated and the result depends on the right operand

If the **left** operand of an `and` operator is False then the right operand is not evaluated

If the **left** operand of an `and` operator is True then both operands are evaluated and the result depends on the right operand



# Short circuiting: or (disjunctions)

```
def is_true() -> bool:  
    print('called is_true()')  
    return True  
  
def is_false() -> bool:  
    print('called is_false()')  
    return False
```

```
is_true() or is_false()
```

called is\_true()

True

is\_false() is **not** called

```
is_false() or is_true()
```

called is\_false()  
called is\_true()

Both is\_false() and  
is\_true() are called

True



# Short circuiting: and (conjunctions)

```
def is_true() -> bool:  
    print('called is_true()')  
    return True  
  
def is_false() -> bool:  
    print('called is_false()')  
    return False
```

```
is_true() and is_false()
```

```
called is_true()  
called is_false()  
  
False
```

Must call both functions to determine if the expression is True/False

```
is_false() and is_true()
```

```
called is_false()  
  
False
```

Only first function must be called to know that the expression is False



# if .. then ... elif ... else statement

```
if condition_1:  
    statement_1_1  
    statement_1_2  
elif condition_2:  
    statement_2_1  
    statement_2_2  
else:  
    statement_3_1  
    statement_3_2
```

If condition\_1 is True, then execute the statements in this block

If condition\_1 is False and condition\_2 is True, then execute the statements in this block

If all previous conditions are false, then execute the statements in this block

0 or more elif statements  
0 or 1 else statements



# Nested if statements

```
home_state: str = 'NJ'  
age: int = 16  
  
if home_state == 'NJ':  
    if age >= 17:  
        print("Eligible for car license")  
    else:  
        print("Maybe next year")  
elif home_state == "IA":  
    if age >= 16:  
        print("Eligible for car license")  
    elif age >= 14:  
        print("Eligible for tractor license")  
    else:  
        print("Maybe next year")
```

Maybe next year

Which else goes with which if?

Indentation defines the statement blocks

Careful indentation is ***critical!***

Python does ***not*** have a case/switch statement

# Your turn

What is the output if  $x == 3$ ?

```
if x <= 4:  
    if x <= 2:  
        if x == 1:  
            print('1')  
        else:  
            print('2')  
    elif x == 3:  
        print('3')  
    else:  
        print('4')  
elif x == 5:  
    print('5')  
else:  
    print('> 5')
```

*McCabe's Cyclomatic complexity*

measures the number of potential paths through if statements.

Empirical studies show that functions with more than about 10 different paths are too hard to maintain.

Keep the number of alterative paths small to improve readability and maintainability.



# Conditional one liners

```
n: int = 42
result1: str = ""

if n >= 0:
    result1 = 'Positive'
else:
    result1 = 'Negative'
```

We can collapse a conditional into a single line

*True\_result if Conditional\_expr else False\_result*

```
result2: str = 'Positive' if n >= 0 else 'Negative'
```

This is known as the **ternary operator**

Code may be more concise, but possibly harder to read



# return – Get me out of here

Functions normally return after the last statement, but you may want to return from the middle of the function

```
def print_square_root(n: int) -> None:  
    if n < 0:  
        print('Only positive numbers, please')  
        return  
  
    result: float = n ** 0.5  
    print(result)
```

Careful with early return from functions

May make functions harder to read and understand

Early return may be best solution in some cases



# return – Don't let this happen to you

```
def absolute_value(n: float) -> float:  
    result: float  
    if n < 0:  
        result = -1 * n  
        return result  
    else:  
        result = n
```

float handles both int  
and float

What's returned if  $n \geq 0$ ?

result?

No! No explicit `return` statement returns `None`

Insure that all exit paths have return statements



# return – Don't forget me!!!

What's wrong with this code?

```
def this_cant_be_good():
    do_something_useful_1()
    return
    do_something_useful_2()
    do_something_useful_3()
```

These statements will  
never be executed

Be alert for unreachable statements

Static analysis tools can help to find these problems



# while loops

while conditional:

    statement<sub>1..n</sub>

statement<sub>n+1</sub>

Execute statement<sub>1..n</sub>  
while conditional is True

```
n: int = 0
while n < 2:
    print(n)
    n += 1
print('done')
```

Verify that the Boolean  
expression eventually  
becomes False to avoid  
infinite loops

0

1

done



# Write a function...

Write a function to calculate the sum of a range of integers

```
def sum_range_while(start: int, end: int) -> int:  
    """ calculate the sum of the integers starting  
        at start and up to, but not including, end  
    """  
  
    total: int = 0  
    while start < end:  
        total += start  
        start += 1  
    return total  
  
  
assert sum_range_while(1,1) == 0  
assert sum_range_while(1,5) == 10  
assert sum_range_while(0,2) == 1
```

What if  $\text{start} \geq \text{end}$ ?

`assert expr` complains if  
expr is not True



# break out

Sometimes you want to break out of the middle or end, rather than the top, of a loop

```
while conditional:  
    statements  
    if condition:  
        break  
    statements|  
statement
```

**break** leaves the loop and execution moves to the first line following the while block



# do while?

Python does **not** include a do while statement...

... but it's easy to simulate

```
while True:  
    statements  
    if condition: # test to terminate loop  
        break  
    statement
```



# for loops

for loops iterate through sequences

```
for item in ['bagel', 'sandwich', 'coffee']:  
    print(f"I'm hungry for a {item}")
```

I'm hungry for a bagel  
I'm hungry for a sandwich  
I'm hungry for a coffee

Python has many different sequences, e.g. lists, dicts, tuples, strings, ranges, generators, ...



# for (i = 0; i < 3; i++){ }

for loops iterate through sequences

What about a range of 0 to  $n$ ?

```
for i in range(3):  
    print(i)
```

0  
1  
2

range(limit) returns a sequence of integers from 0  
**up to, but not including, limit**



# ADVANCED/OPTIONAL

Slides marked **ADVANCED/OPTIONAL**  
contain topics that we'll cover in detail later in the course

```
if comfortable_with_advanced_topics():
    use_in_homework_assignments()
elif confused():
    don't_worry_for_now()
    use_simpler_approach()
```



# Python lists

## ADVANCED/OPTIONAL

Lists hold arbitrary, ordered values of different types

```
from typing import List, Any  
lst: List[Any] = ['coffee', 42]  
lst[0]
```

'coffee'

Create a list

Access individual elements with slices offset from 0

```
lst[0] = 'tea'  
lst[0]
```

'tea'

Change the list with slices

```
lst.append(3.14)
```

Append items to the end of the list

```
for item in lst:  
    print(item)
```

Iterate through the elements in the list

coffee  
42  
3.14



# Type hints for lists ADVANCED/OPTIONAL

Lists hold arbitrary, ordered values of different types

If all elements in the list have the same type, then specify that type in the type hint

Import type hints from typing

```
from typing import List, Any
```

```
primes: List[int] = [1, 2, 3, 5, 7, 11]
```

```
radius: List[float] = [1.1, 2.2, 3.3]
```

```
names: List[str] = ['Nanda', 'Maha', 'Fei']
```

```
values: List[Any] = [1, 2.3, ['a', 4, ['b', 5]]]
```

Type of values in the list



# Finding an item in a collection

**ADVANCED/OPTIONAL**

I need to find my friend Nanda

I can knock on every door until  
I find him ...

... or I can look up his address  
and go directly to the right  
door

Dictionaries map keys to values

Nanda → Apt 5B





# Python dict maps a key to a value

**ADVANCED/OPTIONAL**

Key

Value

```
translate:Dict[str, str] = {'one': 'uno', 'two': 'dos'}  
translate['two']
```

Initialize a dict

'dos'

Use key to retrieve associated value

```
translate['three'] = 'tres'
```

Add new key/value pairs

```
print("English", "Spanish")  
for key, value in translate.items():  
    print(key, "\t", value)
```

English	Spanish
one	uno
two	dos
three	tres

Iterate through key/value pairs

The order is undefined...

# Type hints for dicts

**ADVANCED/OPTIONAL**

Lists hold arbitrary, ordered values of different types

If all elements in the list have the same type, then specify that type in the type hint

var: Dict[key\_type, value\_type]

Type of the key

Type of the value

```
from typing import Dict, Tuple  
  
english2spanish:Dict[str, str] = {'one': 'uno', 'two': 'dos'}  
  
int2str:Dict[int, str] = {1:'one', 2:'two', 3: 'three'}  
  
str2int:Dict[str, int] = {'one': 1, 'two': 2, 'three': 3}  
  
rooms: Dict[Tuple[str, str], str] = {('SSW', '810A'): "GN 204"}
```



# Typical Python Program Files

```
1  """
2      This file demonstrates a typical Python file
3  """
4  from typing import List, Tuple, Dict
5
6  def say_hello(whom: str) -> str:
7      """ return a string of "Hello {whom}!" """
8      return f"Hello {whom}!"
9
10 def main() -> None:
11     """ Say hello to a few friends """
12     print(say_hello("Nanda"))
13     print(say_hello("Maha"))
14
15 if __name__ == "__main__":
16     main()
```

Docstring to describe the file

Imports go at the top of the file

Functions with docstrings and logic

main() function with test cases and user interaction

Call main()



# \_\_name\_\_ identifies current module

**ADVANCED/OPTIONAL**

"\_\_foo\_\_" specifies something magic in Python

file1.py

```
print(f"file1.py: __main__={__name__}")
if __name__ == "__main__":
    print(f"file1.py: running as main")
else:
    print(f"file1.py: being imported")
```

\$ python file1.py

```
file1.py: __main__=__main__
file1.py: running as main
```

\_\_name\_\_ specifies the name of the current module

file2.py

```
print(f"file2.py: __name__={__name__}")

import file1
if __name__ == "__main__":
    print(f"file2.py: running as main")
else:
    print(f"file2.py: being imported")
```

\$ python file2.py

```
file2.py: __name__=__main__
file1.py: __main__='file1'
file1.py: being imported
file2.py: running as main
```



# Is that all?

Python has many more features that we'll explore throughout the semester but this gives you enough to get started on your journey to becoming a Pythonista...





# Coding style guidelines

Code lives forever so make it readable and maintainable

Good comments are critical

Don't explain the obvious but do explain subtleties

Include a comment at the top to explain the problem

Explain major sections

Use meaningful variable names

Use consistent white space/blanks

`var: type = value`

Blank lines between sections

We'll explore more formal guidelines later

**good code**  
is like a  
**good joke**  
*- it needs no explanation*

And now  
for something  
completely different...



# Integrated Development Environments

You're ready to write Python code!

Alternatives for interacting with Python

Type into the Python interpreter

Create a .py file in a text editor

Use an Integrated Development Environment (IDE)

Your choice, but an IDE will be easiest and most efficient after a short learning curve

Many IDEs available

VS Code, PyCharm, Jupyter, PyScripter, IDLE, ...





# Visual Studio Code

The screenshot shows the Visual Studio Code interface with the following components highlighted:

- Stack Trace**: A blue callout points to the bottom-left corner of the interface, which displays the stack trace for the current program state.
- Variable explorer**: A blue callout points to the Variables panel on the left, which shows the argument `n: 1`.
- Code Editor**: A blue callout points to the main code editor area where the `factorial` function is defined.
- Program input/outut**: A blue callout points to the bottom right, showing the status bar with terminal information: Integrated Terminal/Console, Anaconda 2.4.0 (x86\_64) Python 3.5.3, Ln 11, Col 1, Spaces: 4, UTF-8, LF, Python, and a smiley face icon.
- Run your program, step through the code, ...**: A blue callout points to the top right, indicating the debugging capabilities of the IDE.

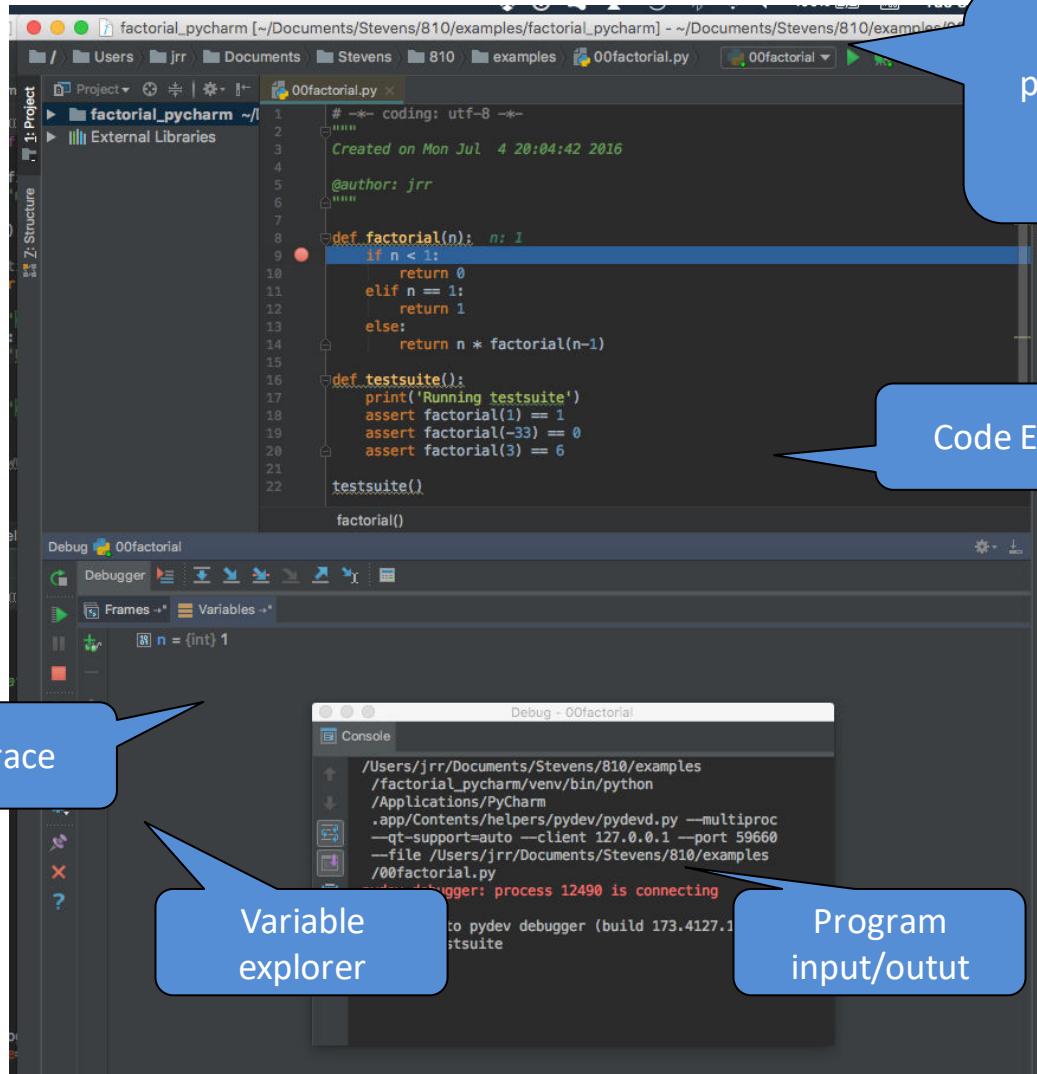
```
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 4 20:04:42 2016
@author: jrr
"""

def factorial(n):
    if n < 1:
        return 0
    elif n == 1:
        return 1
    else:
        return n * factorial(n-1)

def testsuite():
    print('Running testsuite')
    assert factorial(1) == 1
    assert factorial(-33) == 0
    assert factorial(3) == 6

testsuite()
```

# PyCharm



Run your  
program, step  
through the  
code, ...

Code Editor

PyCharm is VERY powerful  
and includes many powerful  
features but may be harder  
to get started

Stack Trace

Variable  
explorer

Program  
input/outut



# Jupyter Notebook

This is a sample notebook from <http://nbviewer.jupyter.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-4-Matplotlib.ipynb>

```
In [1]: from pylab import *
import matplotlib
import matplotlib.pyplot as plt
x = np.linspace(0, 5, 10)
y = x ** 2

In [2]: %matplotlib inline

In [3]: x = np.linspace(0, 5, 10)
y = x ** 2

figure()
plot(x, y, 'r')
xlabel('x')
ylabel('y')
title('title')
show()
```

A plot showing a red curve  $y = x^2$  from  $x=0$  to  $x=5$ .

Notebooks run  
in a browser

Markdown Cell

Code Cell

Jupyter is very popular for Data Science because you can combine code, Markup, output, etc.

Describe the logic and results of the analysis in a single, executable notebook

Graphic output  
from running  
the code cell



# Debugging Jupyter Notebooks

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** jupyter try Last Checkpoint: 2 hours ago (unsaved changes)
- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Help, Cell Toolbar: None
- Code Editor:** The code cell contains Python code for calculating factorials and running a testsuite. A callout points to the line `Tracer()()` with the text "Force a breakpoint".
- Output Area:** The output shows the execution of the code. A callout points to the line `ipdb> print(n)` with the text "Program input/output".
- Text in Output:** Running testsuite  
> <ipython-input-1-3c12d81f5780>(12)factorial()  
11 Tracer()() # this line invokes the Python Debugger  
---> 12 if n < 1:  
13 return 0  
  
ipdb> print(n)  
1  
ipdb> |

Debugging may be less convenient, with command line ipdb/pdb debugger



# IDE Summary

IDE Product	Advantages	Disadvantages
VS Code	Supports many languages, works well with Python	
PyCharm	Very powerful	Steeper learning curve than other IDEs
Jupyter Notebooks	Very popular choice for Data Science, exploration, and sharing results	Debugger is not tightly integrated and requires command line interface

It's ***your*** choice!

Pick one of these, or any other you prefer,  
but **you must** use an IDE for this course



# Optional review sessions

A little confused about anything in the lecture?

Stuck on the homework assignment?

**Optional** online review session Tuesday evenings at 6:00 PM

See Canvas for URL

Be ready to ask questions



# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *More Python Fundamentals: Classes, Exceptions*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from:

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

“Python 3 Object-oriented Programming, Second Edition”, Dusty Philips,

<http://ezproxy.stevens.edu:2155/book/programming/python/9781784398781>

# Today's topics

Objects, classes, and instances

Encapsulation

Class attributes and methods

Single inheritance

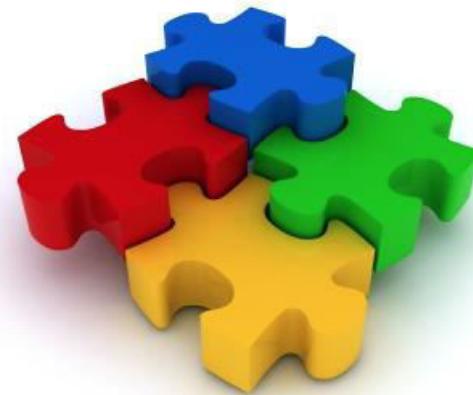
CRC Cards

Exceptions

Catching exceptions

Raising exceptions

Defining custom exceptions





# What problem are we trying to solve?

Writing, testing, reading, and maintaining code is hard

What can we do to help?

Consider a simple example of a rectangle

```
# Describe a rectangle
length: int = 4
width: int = 2
area: int = length * width
perimeter: int = width * 2 + length * 2
print(f"Rectangle: area={area} perimeter={perimeter}")
```

Rectangle: area=8 perimeter=12

That's pretty simple... What's the problem?



# What could possibly go wrong?

That code works great for one instance of rectangle

How does that scale to many rectangles?

What if we add or change features?

```
# Describe a rectangle
length1: int = 4
width1: int = 2
area1: int = length * width
perimeter1: int = width * 2 + length * 2
print(f"Rectangle1: area={area1} perimeter={perimeter1}")
```

Added a new feature (perimeter) to rectangles, but now I need to change code in two places

```
# Describe another rectangle
length2: int = 5
width2: int = 3
area2: int = length2 * width2
perimeter2: int = width1 * 2 + length2 * 2
print(f"Rectangle2: area={area2} perimeter={perimeter2}")
```

Oops! Copy/Paste problem!

```
Rectangle1: area=8 perimeter=12
Rectangle2: area=15 perimeter=14
```

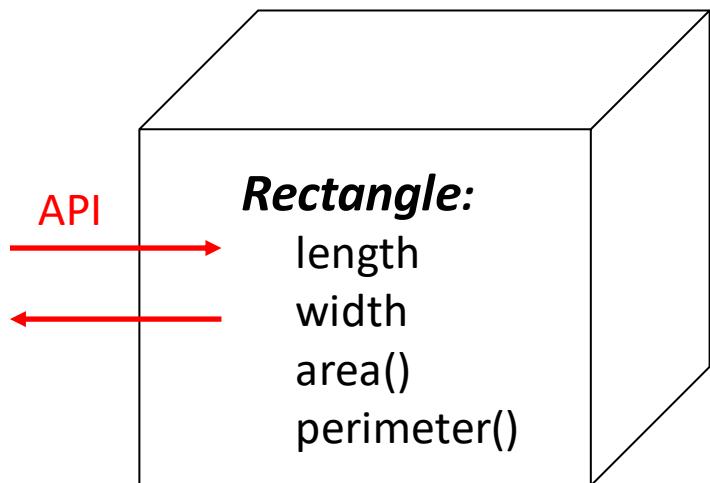
Wait! That should be 16...

# Objects provide encapsulation

Encapsulation bundles data and the operations that work on that data in a ***single place***

Define a “box” that contains everything we need to know about rectangles and reuse that definition many times

Include both data and functions



Now it's easier to create and manipulate many Rectangles  
Easier to isolate Rectangles  
Easier to test, enhance, maintain, and reuse Rectangles  
Everything in one place

# Why objects?

Objects help to design and organize code

Better encapsulation/compartmentalization

Break big problems into small, more manageable chunks

Reduce duplicate code

Increase reusability

Improve readability, maintainability



# Classes represent objects

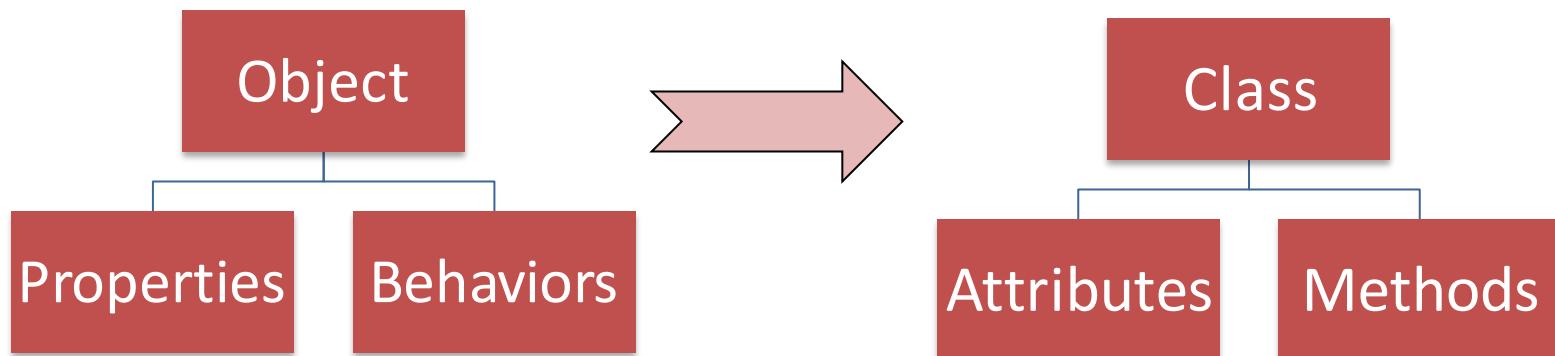
Objects have properties and behaviors

Define classes to represent objects

Classes have attributes and methods

Properties are represented as attributes with values

Methods are represented as functions that perform some behavior for the object





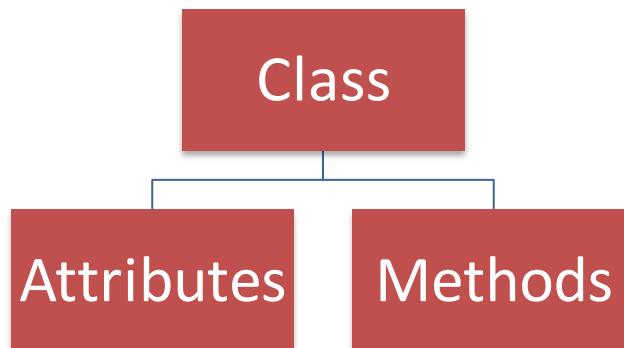
# Object Oriented Programming

Python uses **classes** to represent objects  
Numbers, functions, classes, exceptions, ...

Classes allows defining compound data types

Encapsulation enhances consistency and predictability

**Everything** in Python is an **object**





# Objects, Classes, and Instances

Object

- A physical or logical entity

Class

- The Python representation of an object

Instance

- An instance in memory of a class



# class Rectangle

class names use *CamelCase*

```
class Rectangle:  
    """ Rectangle class """  
    def __init__(self, label: str, length: int, width: int) -> None:  
        self.label: str = label  
        self.length: int = length  
        self.width: int = width  
  
    def area(self) -> int:  
        return self.length * self.width  
  
    def perimeter(self) -> int:  
        return self.length * 2 + self.width * 2  
  
rectangle1 = Rectangle("Rectangle1", 4, 2)  
rectangle2 = Rectangle("Rectangle2", 5, 3)  
print(f"Rectangle1: area={rectangle1.area()} perimeter={rectangle1.perimeter()}")  
print(f"Rectangle2: area={rectangle2.area()} perimeter={rectangle2.perimeter()}")  
  
Rectangle1: area=8 perimeter=12  
Rectangle2: area=15 perimeter=16
```

Initialize the attributes in a new instance of class Rectangle

Methods are functions inside the class definition

Create instances of class Rectangle

instance.method(args) invokes method(self, args) on instance. Python adds the 'self' argument automatically.



# Within a class definition...

## self

Refers to the memory allocated for that instance of the class

First parameter in every method, including `__init__(self)`

## Instance attributes

Data members that are included in every instance of the class

Defined in `__init__(self)`

Accessed as `self.attribute`

Each class instance has its own **distinct copy** of the instance attributes

## Methods

Functions defined for this class that may access any attribute using `instance.method(args)`

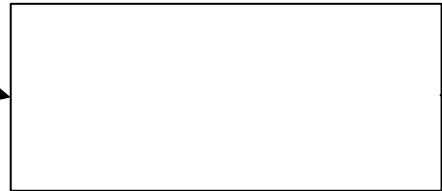
Python passes instance as self

# Create a new instance of class Rectangle

Create a new instance of class Rectangle

```
rectangle1: Rectangle = Rectangle('Rectangle1', 4, 2)
```

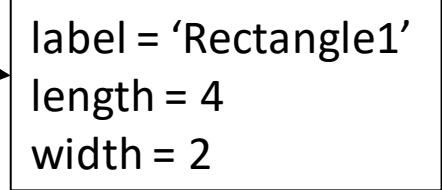
rectangle1



Python allocates space for a new instance of class Rectangle

```
Rectangle.__init__(rectangle1, 'Rectangle1', 4, 2)
```

rectangle1



Python calls Rectangle.\_\_init\_\_() with the new instance and arguments

The new instance is initialized by \_\_init\_\_() and the attributes are defined and initialized



# Inside class Rectangle: `__init__(self, ...)`

```
class Rectangle:  
    """ Rectangle class """  
    def __init__(self, label: str, length: int, width: int) -> None:  
        self.label: str = label  
        self.length: int = length  
        self.width: int = width  
  
    def area(self) -> int:  
        return self.length * self.width  
  
    def perimeter(self) -> int:  
        return self.length * 2 + self.width * 2
```

class names use *CamelCase*

Docstring displayed by  
help(Rectangle)

First parameter must be 'self'.  
Specify other parameters as needed

`self.attribute` defines a new *instance attribute* that is distinct to this instance of the class

self needs no type hint

`__init__()` may have *any* logic, not just defining and initializing instance attributes



# Inside class Rectangle: methods

```
class Rectangle:  
    """ Rectangle class """  
    def __init__(self, label: str, length: int, width: int) -> None:  
        self.label: str = label  
        self.length: int = length  
        self.width: int = width  
  
    def area(self) -> int:  
        return self.length * self.width  
  
    def perimeter(self) -> int:  
        return self.length * 2 + self.width * 2
```

First parameter must be 'self'. Specify other parameters as needed

Methods may use any instance attributes along with other arbitrary logic



# Printing class instances

```
class Rectangle:  
    """ Rectangle class """  
    def __init__(self, label: str, length: int, width: int) -> None:  
        self.label: str = label  
        self.length: int = length  
        self.width: int = width  
  
    def area(self) -> int:  
        return self.length * self.width  
  
    def perimeter(self) -> int:  
        return self.length * 2 + self.width * 2
```

How to encapsulate the information about this instance?

```
rectangle1: Rectangle = Rectangle("Rectangle1", 4, 2)  
print(rectangle1)  
print(f"Rectangle1: area={rectangle1.area()} perimeter={rectangle1.perimeter()}")
```

```
<__main__.Rectangle object at 0x111ac6198>  
Rectangle1: area=8 perimeter=12
```

Printing a class instance prints the context, type, and the memory address



# Printing class instances

`__str__(self)` is a **magic method** that returns a string to describe a specific instance of a class

```
class Rectangle:
    """ Rectangle class """
    def __init__(self, label: str, length: int, width: int) -> None:
        self.label: str = label
        self.length: int = length
        self.width: int = width

    def area(self) -> int:
        return self.length * self.width

    def perimeter(self) -> int:
        return self.length * 2 + self.width * 2

    def __str__(self) -> str:
        return f"{self.label}: area={self.area()} perimeter={self.perimeter()}"
```

Encapsulate relevant information  
about this instance of class Rectangle  
inside the class definition

```
rectangle1: Rectangle = Rectangle("Rectangle1", 4, 2)
print(rectangle1)
```

```
Rectangle1: area=8 perimeter=12
```



# Huh???

# What could possibly go wrong?

```
class Oops:  
    def init(self, n: int): # OOPS! That should be __init__(self, n: int)  
        self.n:int = n  
  
oops: Oops = Oops(3)
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-11-c8a8aeale9c1> in <module>  
      3         self.n:int = n  
      4  
----> 5 oops: Oops = Oops(3)  
  
TypeError: Oops() takes no arguments
```

Check spelling of \_\_init\_\_(self)



# Attributes can be added anytime

By convention, instance attributes are added in `__init__()`

Unlike C++ and Java, Python allows adding attributes to an instance of a class at any time

Including an attribute in `__init__(self)` adds the attribute to all instances

C++ and Java require all attributes to be included in the class definition

This might be a great feature or a unexpected (unpleasant) surprise





# Attributes can be added anytime

```
class College:  
    def __init__(self):  
        """ Class College has no attributes by default """
```

```
Stevens: College = College() # a new instance of class College  
Stevens.mascot = 'Attila'  
Stevens.mascot  
'Attila'
```

The Stevens instance of College now has attribute 'mascot' that was not defined in \_\_init\_\_(self)

```
Columbia: College = College() # another instance of class College  
Columbia.mascot
```

-----  
**AttributeError**  
<ipython-input-14-9415efcdf537> in <module>  
 1 Columbia: College = College() # another instance of class College  
----> 2 Columbia.mascot

Traceback (most recent call last)

**AttributeError:** 'College' object has no attribute 'mascot'



# Avoid accidental attributes

Every class includes a `__slots__` attribute that allows you to restrict accidental attributes

Only attributes included in `__slots__` will be allowed

```
class College:  
    __slots__ = ['mascot'] # predefine and restrict that attributes
```

```
    def __init__(self, mascot: str) -> None  
        self.mascot: str = mascot
```

```
stevens: College = College('Attila')  
stevens.mascot
```

```
'Attila'
```

```
stevens.ascot = "Attila"
```

```
-----  
AttributeError  
<ipython-input-17-32502998a90c> in <module>  
----> 1 stevens.ascot = "Attila"
```

```
AttributeError: 'College' object has no attribute 'ascot'
```

`__slots__` is a class attribute that is shared across all instances of class College

Oops! 'mascot', not 'ascot'



# Attributes guidelines

Initialize **all** class instance attributes in `__init__(self)`

This insures they are defined and initialized

Helps readers to understand which attributes are available

**Beware** of typos that don't generate warnings

Specify `__slots__` to avoid surprises

```
class Foo:  
    def __init__(self) -> None:  
        self.cnt: int = 0
```

No `__slots__` defined so attributes can be added anywhere

```
f: Foo = Foo()  
f.ctn = 2  
f.cnt
```

Oops!! I meant to type 'cnt' but my typo added a new 'ctn' attribute available ONLY to the f instance of class Foo

0



# Type hints in class definitions

Don't annotate 'self'

`__init__()` returns None

```
class Fraction:  
    """ Support addition of fractions """  
    def __init__(self, num: float, denom: float) -> None:  
        """ set num and denom """  
        self.num: float = num  
        self.denom: float = denom  
  
    def __str__(self) -> str:  
        """ String to display fraction """  
  
    def plus(self, other: "Fraction") -> "Fraction":  
        """ Add two fractions and return a new instance of class Fraction """
```

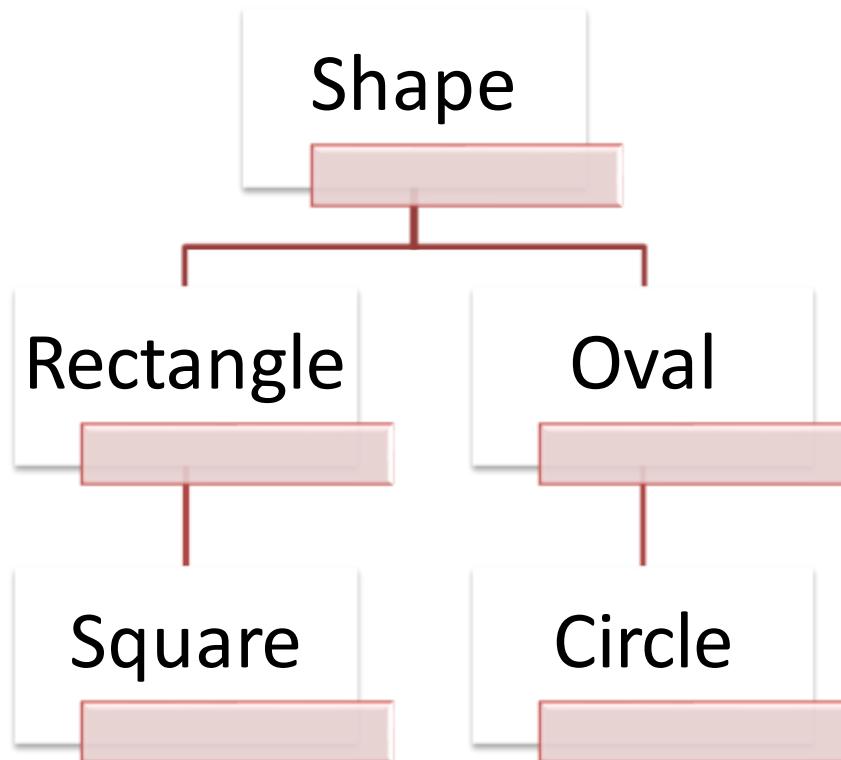
Specify type hint for instance attributes

Use string with class name for class being defined

# Inheritance

## ADVANCED/OPTIONAL

Inheritance provides a convenient mechanism to represent relationships between objects with an IS-A relationship



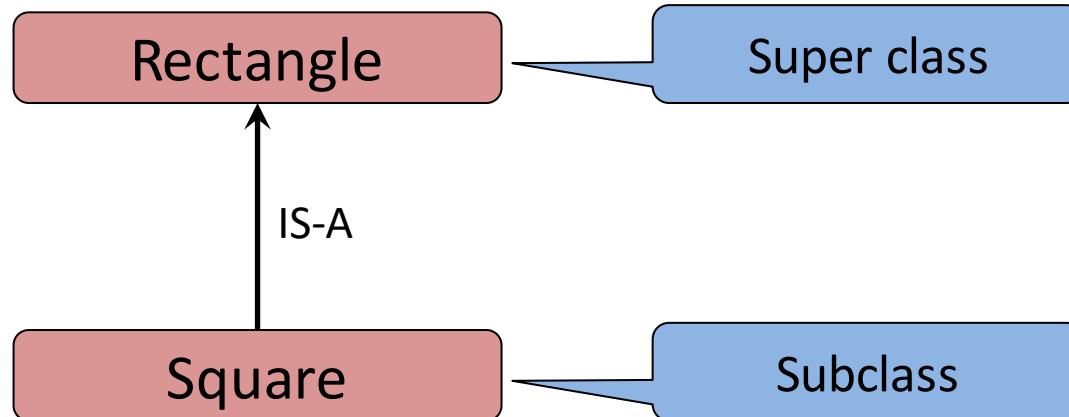
# Single Inheritance

## ADVANCED/OPTIONAL

Inheritance defines a new class

The new class inherits all methods and class attributes

Allows specialization to add new attributes and methods





# class Square

```
class Rectangle:
    """ Rectangle class """
    def __init__(self, label: str, length: int, width: int) -> None:
        self.label: str = label
        self.length: int = length
        self.width: int = width

    def area(self) -> int:
        return self.length * self.width

    def perimeter(self) -> int:
        return self.length * 2 + self.width *

    def __str__(self) -> str:
        return f"{self.label}: area={self.area()} perimeter={self.perimeter()}"
```

class Square(Rectangle):
 """ Square is-a Rectangle. Inherit all methods from Rectangle """

square1: Square = Square("Square1", 4, 4)
print(square1)

Square inherits all methods from Rectangle, including `__init__()`

Wait! That's not convenient... Want a new `Square.__init__(self, label, len)`

Square1: area=16 perimeter=16

Creating a Square calls Rectangle:`__init__()`



# class Square

Redefine Square.\_\_init\_\_()

```
class Square(Rectangle):
    """ Square is-a Rectangle. Inherit all methods from Rectangle """
    def __init__(self, label: str, length: int) -> None:
        print(f"Square.__init__: {label}")

square1: Square = Square("Square1", 4)
print(square1)

Square.__init__: Square1
```

Square inherited \_\_init\_\_(self, label, length, width) from Rectangle but we redefined \_\_init\_\_(self, label, length).

```
AttributeError
<ipython-input-24-db4cd7b56f0d> in <module>
 37
 38     square1: Square = Square("Square1", 4)
--> 39     print(square1)
```

Traceback (most recent call last)

Oops! Redefining \_\_init\_\_() lost all of the instance attributes defined in Rectangle.\_\_init\_\_()

```
<ipython-input-24-db4cd7b56f0d> in __str__(self)
 29
 30     def __str__(self) -> str:
--> 31         return f"{self.label}: area={sel:
 32
 33 class Square(Rectangle):
```

We **could** include all of the instance attributes from the superclass in \_\_init\_\_() but what if the definition of Rectangle changes?

```
AttributeError: 'Square' object has no attribute 'label'
```

# Overriding `__init__()` ADVANCED/OPTIONAL

Any attribute or method from the superclass can be overridden in the subclass, including `__init__()`

Overriding a method **replaces** the definition of the method in the subclass

**Careful!** If you override `__init__()` then the superclass's `__init__()` is **NOT** called (unless you call it explicitly)

C++ automatically invokes the parent's constructor  
`__init__()` typically defines the instance attributes so those instance attributes won't be included in the subclass

Instance attributes are not inherited because they are not part of the class definition. They are created by `__init__()`





# class Square: redefine \_\_init\_\_()

```
class Rectangle:  
    """ Rectangle class """  
    def __init__(self, label: str, length: int, width: int) -> None:  
        self.label: str = label  
        self.length: int = length  
        self.width: int = width  
  
    def area(self) -> int:  
        return self.length * self.width  
  
    def perimeter(self) -> int:  
        return self.length * 2 + self.width * 2  
  
    def __str__(self) -> str:  
        return f"{self.label}: area={self.area()} perimeter={self.perimeter()}"
```

Delegate initializing the superclass to the subclass.  
Don't replicate the code in the subclass

Call super().\_\_init\_\_() from \_\_init\_\_() to initialize superclass instance attributes

```
class Square(Rectangle):  
    """ Square is-a Rectangle. Inherit all methods from Rectangle """  
    def __init__(self, label: str, length: int) -> None:  
        print(f"Square.__init__: {label}")  
        super().__init__(label, length)
```

Rectangle.\_\_init\_\_() is invoked in call to super().\_\_init\_\_()

```
square1: Square = Square("Square1", 4)  
print(square1)
```

All class and instance attributes from Rectangle are available in Square after calling super().\_\_init\_\_()

```
Square.__init__: Square1  
Square1: area=16 perimeter=16
```



# Single Inheritance: Employee

## Class Employee:

- Attributes

`name` —————

`title` —————

- Methods

`__init__()` —————

`work()` —————

`__str__()` —————

## Class Manager:

- Attributes

`name`

`title`

`team_size`

- Methods

`__init__()`

`work()`

`__str__()`

Class Manager inherits all of class Employee's methods, including `__init__()`.

Add new attributes or replace method behavior

## Manager *is-a* Employee

Managers have an additional attribute: `team_size`

Managers `work()` method is different from Employee's

Managers have the same `__str__()` as Employees

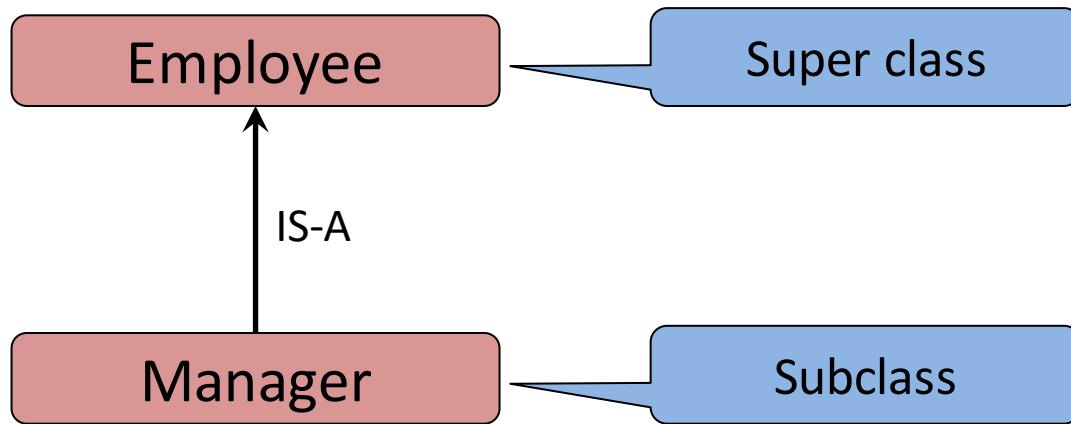
# class Employee

```
class Employee:  
    job = {'developer': 'developing', 'tester': 'testing'}  
  
    def __init__(self, name: str, title: str) -> None:  
        self.name: str = name  
        self.title: str = title  
  
    def work(self) -> None:  
        print(f"{self.name} is busy {Employee.job[self.title]}")  
  
    def __str__(self) -> str:  
        return f"<Employee: {self.name} {self.title}>"  
  
nanda: Employee = Employee('Nanda', 'developer')  
print(nanda)  
nanda.work()  
sujit: Employee = Employee('Sujit', 'tester')  
print(sujit)  
sujit.work()
```

```
<Employee: Nanda developer>  
Nanda is busy developing  
<Employee: Sujit tester>  
Sujit is busy testing
```

Class attributes are defined in the class and are shared across all instances of the class

# Single Inheritance





# Overriding methods **ADVANCED/OPTIONAL**

A subclass may override a method of the super class

Python chooses the appropriate method to invoke at run time based upon the type of the instance

```
class Employee:
    job = {'developer': 'developing', 'tester': 'testing'}

    def __init__(self, name: str, title: str) -> None:
        self.name: str = name
        self.title: str = title

    def work(self) -> None:
        print(f"{self.name} is busy {Employee.job[self.title]}")

    def __str__(self) -> str:
        return f"<Employee: {self.name} {self.title}>"

class Manager(Employee):
    def __init__(self, name: str, sz: int) -> None:
        super().__init__(name, "manager") # call Employee.__init__()
        self.team_size: int = sz # unique to Manager

    def work(self) -> None:
        print(f"{self.name} is managing a team of {self.team_size}")

    # inherit __str__() from Employee

jim: Manager = Manager('Jim', 20)
jim.work()
print(jim)
```

work() is defined in both Employee and Manager

Manager.work() replaces Employee.work() in instances of class Manager

```
Jim is managing a team of 20
<Employee: Jim manager>
```



# class Manager

```
class Employee:
    job = {'developer': 'developing', 'tester': 'testing'}

    def __init__(self, name: str, title: str) -> None:
        self.name: str = name
        self.title: str = title

    def work(self) -> None:
        print(f"{self.name} is busy {Employee.job[self.title]}")

    def __str__(self) -> str:
        return f"<Employee: {self.name} {self.title}>"

class Manager(Employee):
    def __init__(self, name: str, sz: int) -> None:
        super().__init__(name, "manager") # call Employee.__init__()
        self.team_size: int = sz # unique to Manager

    def work(self) -> None:
        print(f"{self.name} is managing a team of {self.team_size}")

    # inherit __str__() from Employee

jim: Manager = Manager('Jim', 20)
jim.work()
print(jim)

Jim is managing a team of 20
<Employee: Jim manager>
```

Manager is-a Employee

Manager inherits all methods and class attributes from Employee, including `__init__()` which initializes instance attributes. We can also add new attributes and methods to the subclass.



# Overriding Superclass' `__init__()`

```
class Employee:
    job = {'developer': 'developing', 'tester': 'testing'}

    def __init__(self, name: str, title: str) -> None: ...

    def work(self) -> None: ...
    def __str__(self) -> str: ...

class Manager(Employee):
    def __init__(self, name: str, sz: int) -> None:
        super().__init__(name, "manager") # call Employee.__init__()
        self.team_size: int = sz # unique to Manager

    def work(self) -> None: ...
    |
# inherit __str__() from Employee
```

Manager.`__init__()` replaces Employee.`__init__()` so Employee's instance attributes are not defined in Manager unless we call explicitly `super().__init__()`



# Single Inheritance Summary

The subclass inherits all **methods** from the superclass

Super class instance attributes may be available, depending on if  
`__init__()` is redefined in the subclass or if `super().__init__()` is called

The subclass may add new class attributes, instance attributes, and methods

Any method, including `__init__()`, can be overridden by the subclass

If you override `__init__()` then you may want to call `super().__init__()` to create and initialize the superclass's instance variables in the subclass

Or, define instance attributes needed by the subclass but that may defeat the advantages of using inheritance.

# When to use OOP?

Python supports both *Procedural* and *Object Oriented Programming*  
When to use each approach?

Objects have **both** properties and behaviors

If only behaviors then a function may be simpler

If only properties, then a Python data structure may be better

If both, then a class is a good solution

How many instances of an object do you need?

Only one? A class may be overkill

Many? Class is a great solution



If inheritance is natural, then classes are a great solution

Use classes anytime they improve readability!

# When to use OOP?

**Task:** write a program to summarize a file with the number of lines, words, and characters

Should we define a class?

Might we need to represent multiple files?

Class instances make that trivial

```
students = FileReader('students.txt')  
majors = FileReader('majors.txt')
```



Encapsulation makes the code easier to understand, maintain, and reuse

# Object Oriented Design

You're working on a new problem...

A small gray 3D humanoid figure stands next to a large red question mark, appearing to be in thought or confusion.

What objects  
should I define?

What are the  
attributes and  
methods?

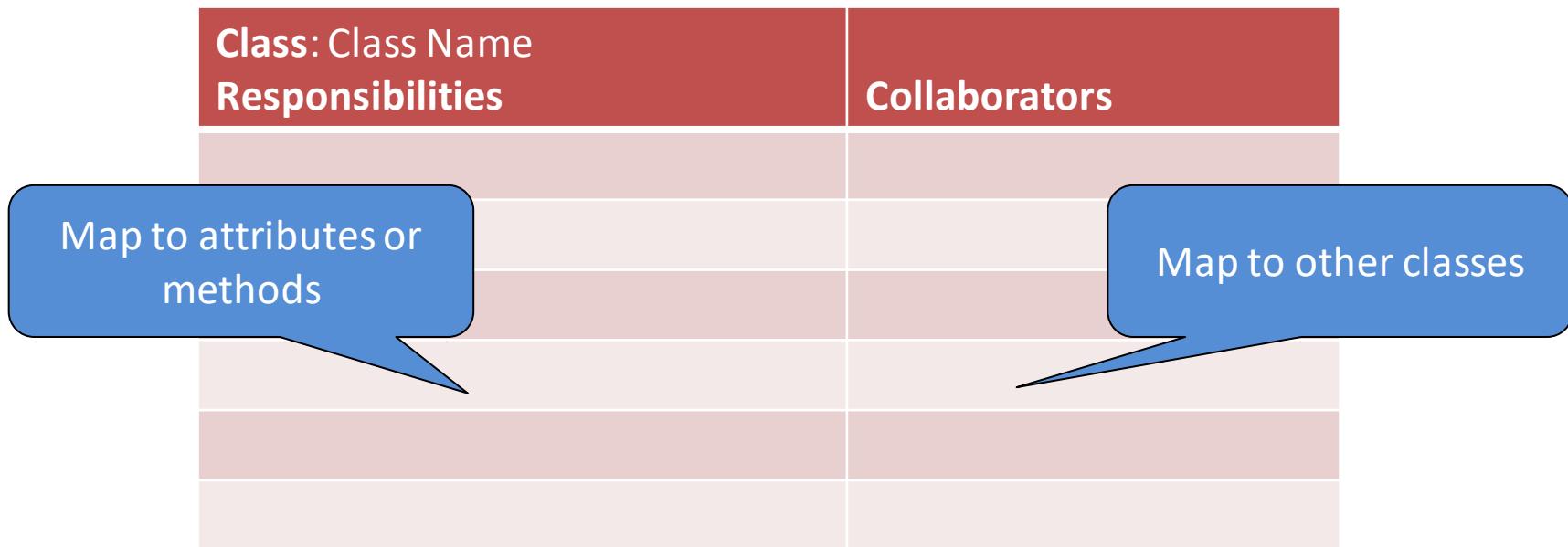


# CRC Cards

**Class:** name of the class

**Responsibilities:** what must the class do?

**Collaborators:** other classes that will help





# CRC Cards Example

Create CRC cards for the Rock/Paper/Scissors game

1. Identify a class (frequently nouns in the domain)
2. Add responsibilities (verbs associated with classes)
3. For each responsibility, identify collaborators
4. Create new classes for new collaborators
5. Iterate as you detect missing classes or collaborators

Class: Game Responsibilities	Collaborators	Class: Player Responsibilities	Collaborators
Get human move	Player	Choose move	
Get computer move	Player		
Identify winner			



# Microsoft Design Guidelines

## Separation of Concerns

- Minimize overlap between components

## Single Responsibility Principle

- A class/method/function should do one thing well

## Principle of Least Knowledge

- Minimize what others need to know about your internals (don't know and don't care)

## Don't Repeat Yourself (DRY)

- Don't duplicate functionality in multiple places

## Minimize Upfront Design (KISS)

- Keep it simple and elegant

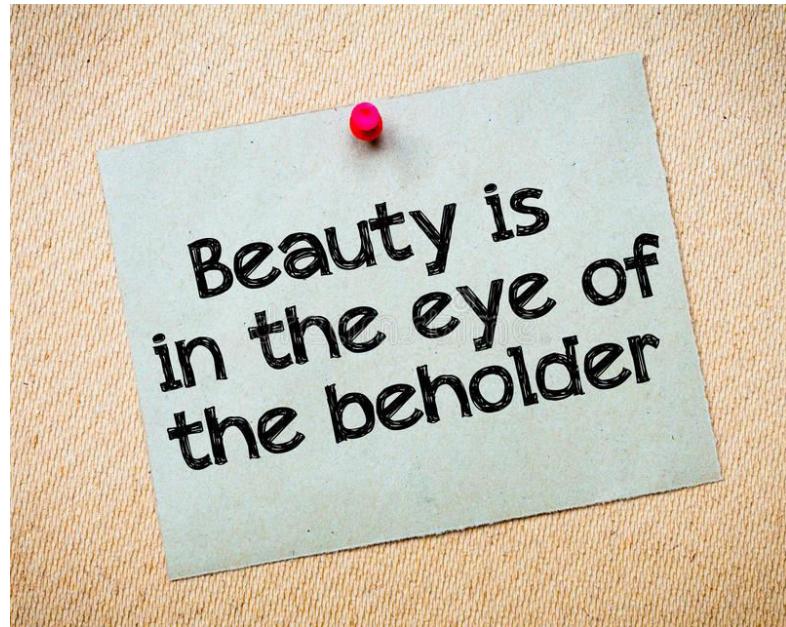
<https://msdn.microsoft.com/en-us/library/ee658124.aspx>

# Design Summary

Good design is a thing of beauty

There are guidelines and heuristics but no cookbook

Good design is learned through study, experience and review



And now  
for something  
completely different...





# Things don't always go as planned...

3 / 0

**ZeroDivisionError: division by zero**

```
int('zero')
```

**ValueError: invalid literal for int() with base 10: 'zero'**

```
open("file doesn't exit")
```

**FileNotFoundException: [Errno 2] No such file or directory: "file doesn't exist"**

These are all examples of Python **exceptions**



# Exceptions: anticipate problems

```
try:  
    # risky operation  
except exception_type_1:  
    # do this if exception_type_1 occurs during risky operation  
except exception_type_2:  
    # do this if exception_type_2 occurs during risky operation  
else:  
    # do this if no exception occurred  
finally:  
    # Do this after all exception processing  
    # Frequently used to free resources  
    # This block is executed whether an exception was raised or not|
```



# Anticipate problems with user input

```
inp: str = input("Enter an integer between 1 and 20:")
try:
    n: int = int(inp)
except ValueError:
    print(f"{inp} is not an integer")
else:
    print(f"{n} + 1 == {n + 1}")
```

```
Enter an integer between 1 and 20:3
3 + 1 == 4
```



# Anticipate problems opening a file

```
from typing import IO

file_name: str = input("Enter file name: ")
try:
    fp: IO = open(file_name, 'r')
except FileNotFoundError as e:
    print(f"Can't open '{file_name}' ({e})")
else:
    with fp:
        for line in fp: # fp moves line by line
            # process the line
            print(line)
```

'e' describes the exception

str(e) displays an error message associated with the exception

Enter file name: not a file

Can't open 'not a file' ([Errno 2] No such file or directory: 'not a file')



# Common Python Built In Exceptions

Exception Name	Description
Exception	Base class for all exceptions
ValueError	Raised when an invalid value is seen
FileNotFoundException	Raised when attempting to open a non-existent file
StopIteration	Used by generators to stop iteration
SystemExit	Raised by <code>system.exit()</code>
ArithmaticError	Base class for exceptions during numeric calculation
ZeroDivisionError	Raised on division by zero
AssertionError	Raised when an <code>Assert</code> statement fails
AttributeError	Raised when accessing a non-existent attribute
KeyboardInterrupt	Raised when user raises an interrupt, e.g. <code>Ctrl-C</code>
NameError	Raised when an unknown name is accessed
SyntaxError	Raised when Python encounters a syntax error
Your Exception	You can define and raise your own exceptions



# Raising exceptions

Raise an exception in your code to identify error conditions

```
# something is wrong -- let the user know  
raise Exception
```

---

**Exception:**

But which exception?

Use an existing Python exception or build your own with inheritance

```
raise ValueError("Oops! Something is wrong")
```

---

**ValueError: Oops! Something is wrong**



# Unqualified except

Avoid unqualified except, e.g.

```
inp = input("Enter an integer between 1 and 20:")
try:
    n = int(inp)
except: # DON'T USE UNQUALIFIED except STATEMENTS!!!
    print(f"{inp} is not an integer")
else:
    print(f"{n} + 1 == {n + 1}")
```

The Most Diabolical  
Python AntiPattern!!!

This block is invoked for  
ANY exception, not just the  
ones you expect

```
Enter an integer between 1 and 20:1
1 + 1 == 2
```

Unlike other languages with exceptions, Python uses exceptions for many situations



# What was the name of that exception?

Recall, when in doubt, try it out...

What's the name of the exception for divide by 0?

```
3 / 0
```

-----  
`ZeroDivisionError: division by zero`

Here's the name of the exception to  
use in the try/except block

```
try:  
    quotient = num / denom  
except ZeroDivisionError:  
    print("Oops! Divide by 0")
```



# Keep try blocks small

Keep try blocks as small as possible to isolate issues

Exceptions can occur for many reasons

Minimize the lines between the event that may raise an exception and the exception handler

```
# DON'T DO THIS
try:
    # many lines of code
    # risky operation
    # more code
    # more risky operations
except ExceptionType:
    # handle the problem
```

Which code caused the exception to be raised?



# Keep try blocks small

```
# DO THIS INSTEAD
# lines of code
try:
    # risky operation
except ExceptionType:
    # handle the problem
else:
    # more code
try:
    # risky operation
except ExceptionType:
    # handle the problem
else:
    # more code
```

Now you can isolate the reason  
that the exception was raised



# What could possibly go wrong?

```
def sqrt(n: int) -> float:  
    if n < 0:  
        raise ValueError("sqrt of negative number")  
    return n ** 0.5
```

Warn users about negative values

```
try:  
    val = input("Please enter a number:")  
    num = float(val)  
    print(sqrt(num))  
except ValueError:  
    print(f"Expected a number but got: {val}")
```

Warn users about invalid input values

```
Please enter a number:4  
2.0
```

So far, so good...



# What could possibly go wrong?

```
def sqrt(n: int) -> float:  
    if n < 0:  
        raise ValueError("sqrt of negative number")  
    return n ** 0.5  
  
try:  
    val = input("Please enter a number:")  
    num = float(val)  
    print(sqrt(num))  
except ValueError:  
    print(f"Expected a number but got: {val}")
```

Please enter a number:four

Try an invalid input value...

Expected a number but got: four

So far so good...



# What could possibly go wrong?

```
def sqrt(n: int) -> float:  
    if n < 0:  
        raise ValueError("sqrt of negative number")  
    return n ** 0.5  
  
try:  
    val = input("Please enter a number:")  
    num = float(val)  
    print(sqrt(num))  
except ValueError:  
    print(f"Expected a number but got: {val}")
```

Please enter a number:-4

Expected a number but got: -4

WHAT??? -4 is a number!!!



# What could possibly go wrong?

```
def sqrt(n: int) -> float:  
    if n < 0:  
        raise ValueError("sqrt of negative number")  
    return n ** 0.5  
  
try:  
    val = input("Please enter a number:")  
    num = float(val)  
    print(sqrt(num))  
except ValueError:  
    print(f"Expected a number but got: {val}")
```

float(val) raises ValueError on error but so does sqrt(n). Main() doesn't distinguish between the two cases

Please enter a number:-4  
Expected a number but got: -4

WHAT??? -4 is a number!!!



# What could possibly go wrong?

```
def sqrt(n: int) -> float:  
    if n < 0:  
        raise ValueError("sqrt of negative number")  
    return n ** 0.5  
  
try:  
    val = input("Please enter a number:")  
    num = float(val)  
    print(sqrt(num))  
except ValueError:  
    print(f"Expected a number but got: {val}")
```

Limit try/except blocks to just the risky code

```
Please enter a number:-4  
Expected a number but got: -4
```



# Program flow with exceptions

Raising an exception pops the runtime call stack, returning from functions, until the exception is caught:

- in a try/except block somewhere in the call stack
- by the Python interpreter (REPL)

```
def sqrt(n: int) -> float:  
    if n < 0:  
        raise ValueError("sqrt of negative number")  
    return n ** 0.5
```

```
sqrt(-4)
```

Exception caught by the Python interpreter

```
ValueError: sqrt of negative number
```



# Program flow with no exceptions

```
def sqrt(n: int) -> None:
    if n < 0:
        raise ValueError("sqrt of negative number")
    print(f"sqrt({n}) == {n ** 0.5}")

def inner(n: int) -> None:
    print("inner() calling sqrt()")
    sqrt(n)
    print("leaving inner()")

def outer(n: int) -> None:
    try:
        print("outer() calling inner()")
        inner(n)
    except ValueError as e:
        print("Caught exception in outer", e)
    print("leaving outer()")

outer(4)
```

```
outer() calling inner()
inner() calling sqrt()
sqrt(4) == 2.0
leaving inner()
leaving outer()
```

REPL calls outer(4)  
outer(4) calls inner(4)  
inner(4) calls sqrt(4)  
sqrt(4) prints message, returns to inner()  
Inner returns to outer()  
outer() returns to Python REPL



# Program flow with exceptions

```
def sqrt(n: int) -> None:
    if n < 0:
        raise ValueError("sqrt of negative number")
    print(f"sqrt({n}) == {n ** 0.5}")

def inner(n: int) -> None:
    print("inner() calling sqrt()")
    sqrt(n)
    print("leaving inner()")

def outer(n: int) -> None:
    try:
        print("outer() calling inner()")
        inner(n)
    except ValueError as e:
        print("Caught exception in outer",
              file=e)
    print("leaving outer()")

outer(-4)
```

```
outer() calling inner()
inner() calling sqrt()
Caught exception in outer sqrt of negative number
leaving outer()
```

REPL calls outer(-4)  
outer(-4) calls inner(-4)  
inner(-4) calls sqrt(-4)  
sqrt(-4) raises a ValueErrorException  
Inner(-4) doesn't catch the exception  
so execution flows to outer(-4) where  
exception is caught  
Execution continues in outer()  
following except



# Define your own exceptions

```
class NegativeInputError(Exception):
    """ Define a user defined exception """

def check_positive(value: int) -> int:
    if value < 0:
        msg = f"Found negative value: {value}"
        raise NegativeInputError(msg)
    else:
        return value

check_positive(-3)
```

NegativeInputError can be used anywhere you use exceptions

NegativeInputError: Found negative value: -3

# LBYL vs EAFP Strategies



## **LBYL:** Look Before You Leap (C++, Java)

Check carefully before attempting something risky to avoid exceptions

## **EAFP:** Easier to Ask Forgiveness than Permission<sup>1</sup> (Python)

Write maintainable code that assumes everything is okay

Handle the situations where something goes wrong

Leads to more readable and maintainable code



<sup>1</sup>Attributed to Admiral Grace Hopper, a computing pioneer



# LBYL vs EAFP

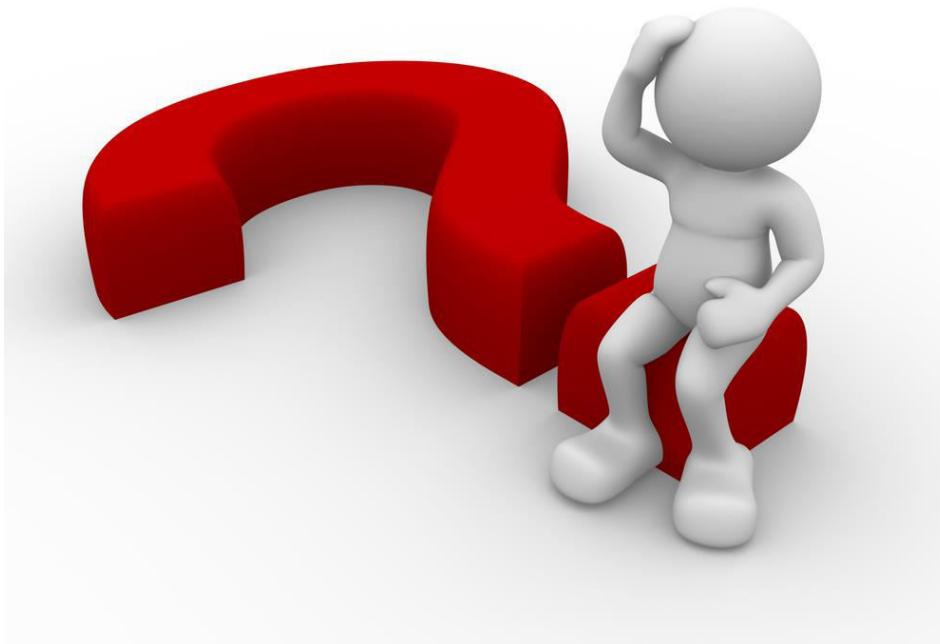
## LBYL: (Java, C++)

```
double  
safe_div(float n, float d) {  
    if (d == 0) {  
        cerr << "divide by 0";  
        return 0;  
    } else {  
        return n/d;  
    }  
}
```

## EAFP: (Python)

```
def safe_div(n, d):  
    try:  
        return n/d  
    except ZeroDivisionError:  
        print("divide by 0")
```

# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Testing and Debugging*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

“Python Testing Cookbook”, Greg L. Turnquist, **ISBN-13:** 978-1849514668

<https://docs.python.org/dev/library/unittest.html>

# Today's topics

## Testing

Test Driven Development

Testing triangle classification

Unittest framework

## Debugging functions

Debugging with print

Debugging with VS Code



# Test Driven Development

Write tests before you write the code

Test Driven Development (Kent Beck)

eXtreme Programming Agile Method

Writing the test helps you to think about the problem and write better code



[https://assertselenium.files.wordpress.com/2012/11/tdd\\_cycle.jpeg](https://assertselenium.files.wordpress.com/2012/11/tdd_cycle.jpeg)

# Software testing

The goal of software testing is to determine if the implementation meets its specifications

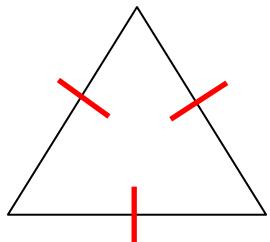
Does it do what it's supposed to do?

Testing and debugging are related, but different tasks

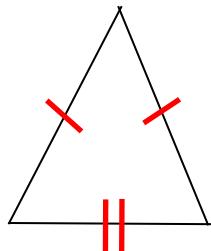
Testing identifies problems – debugging fixes problems



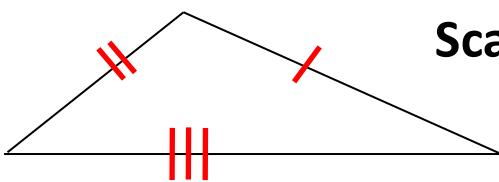
# Testing triangle classification



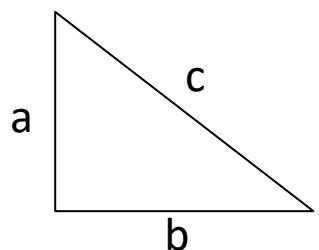
**Equilateral:** all three sides have the same length



**Isosceles:** exactly two sides have the same length



**Scalene:** sides have the three different lengths



**Right:**  $a^2 + b^2 = c^2$



# Requirements

“Write a function `classify_triangle()` that takes three parameters:  $a$ ,  $b$ ,  $c$  representing the lengths of the sides of a triangle. Return a string that specifies whether the triangle is scalene, isosceles, or equilateral, and whether it is a right triangle as well.”

Are these good requirements?

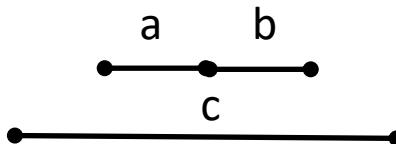
Are they complete?

What's missing from the requirements?

# Test the Requirements

“Write a function `classify_triangle()` that takes three parameters:  $a$ ,  $b$ ,  $c$  representing the lengths of the sides of a triangle. Return a string that specifies whether the triangle is scalene, isosceles, or equilateral, and whether it is a right triangle as well.”

- Define the return values – important for code reuse with APIs
- Define boundary conditions: min/max values, e.g.  $> 0$
- Resolve precedence ambiguity with right vs. scalene triangles
  - Every right triangle is also a scalene triangle
- Include definition of a valid triangle:
  - Sum of any two sides  $<$  third side
- The sides may be specified in any order
  - Don’t assume *side, side, hypotenuse* order
- Other improvements?





# When are we done testing?

What kinds of tests must we include?

How many tests must we include?

How do we know when we've adequately tested the function?

**ARE WE  
DONE  
YET?**

# Testing Strategies

Complete coverage, i.e. test all possible combinations?

Might be possible with a small range but typically not feasible

Choose a subset of values to test

Test each feature at least once

Test expected case

Test “around the edges”

What might we have missed?

Include both positive and negative tests

Positive: valid input produces valid output

Negative: invalid input is handled gracefully

Beware that tests can have bugs too!



# What should we test?

Valid & invalid input parameters

Negative numbers, floats, strings, ...

Boundary conditions

E.g. min value, max value

Equivalence classes

There are many integers and floats

We can't possibly test all of them

Test each possible output case

Test parameter order invariance

```
classify_triangle(3,4,5) == classify_triangle(5,3,4)
```



# Boundary conditions

Example: an integer  $x$  is valid from  $a$  to  $b$



Important test cases occur...	Test cases
<b>At</b> the boundary	$a, b$
A little <b>before</b> the boundary	$a - 1$
A little <b>after</b> the boundary	$b + 1$

Check  $<$  vs  $<=$  and  $>$  vs  $>=$  carefully!

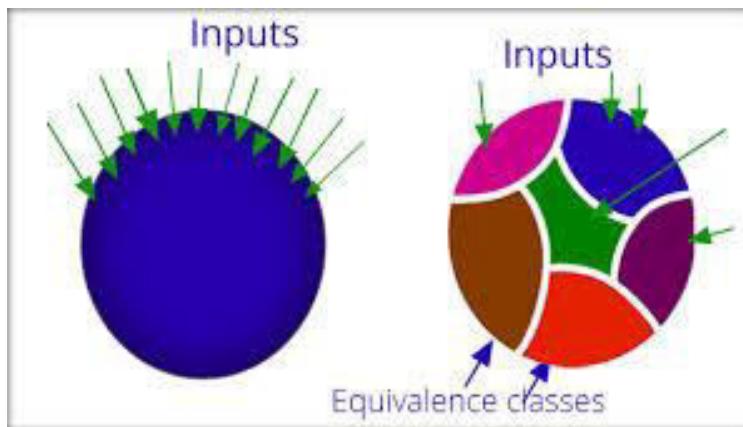
# Equivalence classes

We can't test all possible values

e.g. all float values between 0 and 1

Partition the range of values into equivalence classes, i.e. sets that we expect to behavior similarly

Use one value from the equivalence class to test all values from the equivalence class



# Equivalence classes

Example: an integer  $x$  is valid from  $a$  to  $b$



## Equivalence Classes

- Valid:  $a \leq x \leq b$
- Invalid:  $x < a$ ;  $x > b$
- Invalid:  $x$  not an integer

## Equivalence Class test cases

- Valid value from equivalence set, e.g.  $a$  or  $b$
- Invalid too small
- Invalid too big
- Some non-integers (\$, !@, A)



# Boundary conditions for classify\_triangle

## Boundary conditions

- $a, b, c$  must be numbers (integers or floats)
- $a > 0, b > 0, c > 0$
- Upper limits?
- Sum of any two sides  $>$  third side  $\rightarrow$  Invalid triangle



# Equivalence classes for classify\_triangle

Case	Equivalence class
Valid inputs	Integers/floats > 0
Invalid inputs	Strings, integers/floats <= 0
Equilateral, Isosceles, Scalene	Any 3 numbers, > 0, all three the same
Equilateral, Isosceles, Scalene	Any 3 numbers, > 0, two the same, one different
Equilateral, Isosceles, Scalene	Any 3 numbers, > 0, all three different
Right/Scalene	Any 3 numbers, > 0, $a^2 + b^2 = c^2$
Right/Isosceles	Any 2 numbers, > 0, $a^2 + a^2 = c^2$



# Test cases for classify\_triangle

Test	Inputs	Expected output
Invalid inputs	'hello', 1, 1	Error
	-1, -1, -1	Error
	0, 0, 0	Error
	1, 1, 3	Not a triangle
Equilateral	3.0, 3.0, 3.0	Equilateral
Isosceles	2, 2, 3	Isosceles
Isosceles/Right	2, 2, 2.828 (sqrt(8))	Isosceles Right
Scalene	3, 4, 6	Scalene
Scalene/Right	3, 4, 5	Scalene Right

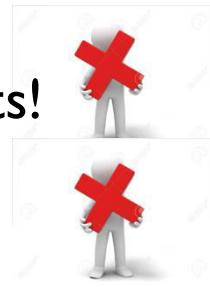
Note: the order of the arguments to classify\_triangle should not change the result but it is important to verify.  
Repeat all of the valid input tests with the same inputs in different orders.

# How to implement and run the tests?

We've defined the tests we need for classifying triangles

How should we run the tests?

I'm a hacker! I don't need tests!



Manually?

Not feasible in changing production environment

Automated testing is critical



Even for homework assignments!



# Unittest Overview

xUnit is a popular framework for software testing

pyUnit/unittest, jUnit, C Unit, C++ unit, ...

Write test cases along with (before) your code

Invoke the test cases

Define Test Harnesses and Test Cases

Unittest manages running the tests

Use inheritance to derive a test case from unittest

Test a class or procedural code



# Simple Unittest recipe



1. import unittest
2. Derive a class `fooTest` from `unittest.TestCase` for the class/feature `foo` being tested (the test class name is arbitrary)
3. Define a set of methods inside `fooTest` for each test case

Each test case includes calls to `unittest.TestCase.assert*` () methods

Method name should begin with '`test_`'
4. Call `unittest.main()`

`unittest.main()` automatically invokes all of the methods in all classes derived from `unittest.TestCase`
5. Debug and fix bugs in test cases and code
6. Repeat until all tests pass



# Unittest Assert Methods

Method	Checks
assertEqual(a, b, msg=None)	a == b
assertNotEqual(a, b, msg=None)	a != b
assertAlmostEqual(a, b, places=7,msg=None)	round(a-b, places) == 0
assertNotAlmostEqual(a, b, places=7,msg=None)	round(a-b, places) != 0
assertTrue(v, msg=None)	bool(v) is True
assertFalse(v, msg=None)	bool(v) is False
assertIs(a, b, msg=None)	a is b
assertIsNot(a, b, msg=None)	a is not b
assertIsNone(v, msg=None)	v is None
assertIsNotNone(v, msg=None)	v is not None
assertIn(a, b, msg=None)	a in b
assertNotIn(a, b, msg=None)	a not in b
assertIsInstance(a, b, msg=None)	isinstance(a, b)
assertNotIsInstance(a, b, msg=None)	not isinstance(a, b)
assertRaises(Exception, function, [function args])	Exception is raised



# Buggy Triangle Code

```
import unittest
class BuggyTriangle:
    def __init__(self, s1: float, s2: float, s3: float) -> None:
        self.a: int = s1
        self.b: int = s2
        self.c: int = s3

    def right_triangle(self) -> bool:
        return round(((self.a ** 2) + (self.b ** 2)), 2) == \
            round((self.c ** 2), 2)
```

Make unittest available to the program

Define the new class



# BuggyTriangle Test Code

```
class BuggyTriangleTest(unittest.TestCase):
    def test_init(self) -> None:
        """ Sides stored properly in __init__() """
        t: BuggyTriangle = BuggyTriangle(3, 4, 5)
        self.assertEqual(t.a, 3)
        self.assertEqual(t.b, 4)
        self.assertEqual(t.c, 5)

    def test_right_triangle(self) -> None:
        """ test right triangle detection """
        t: BuggyTriangle = BuggyTriangle(3, 4, 5)
        self.assertTrue(t.right_triangle())
        self.assertTrue(BuggyTriangle(5, 4, 3).right_triangle())
        self.assertFalse(BuggyTriangle(3, 3, 3).right_triangle())

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)
```

Method name must begin with 'test\_'

Derive a class from unittest.TestCase

Docstring is important for debugging

Test method can include any code plus assert\* methods

Add as many test cases as needed

Run the test cases

Specify verbosity of test output



# BuggyTriangle test output

```
=====
FAIL: test_right_triangle (__main__.BuggyTriangleTest)
test right triangle detection
```

Docstring from the test case

```
Traceback (most recent call last):
```

```
  File "<ipython-input-6-f4288b3be966>", line 13, in test_right_triangle
    self.assertTrue(BuggyTriangle(5, 4, 3).right_triangle())
```

```
AssertionError: False is not true
```

Click on the line to go directly to the code

```
Ran 2 tests in 0.002s
```

Ran 2 tests, 1 failed

```
FAILED (failures=1)
```

A test is one method with  
arbitrary number of assert\*  
statements





# Comparing floats

`assertAlmostEqual(a, b, places) ???`

Be aware that computers don't perfectly represent floating point numbers

```
8 == (8 ** 0.5) ** 2 # 8 == sqrt(8)2
```

False

WHAT???

```
(8 ** 0.5) ** 2
```

8.000000000000002

round(n, places) eliminates precision  
but leads to expected results.  
Be careful with the level of precision  
that is appropriate for your application

```
round(8, 2) == round(((8 ** 0.5) ** 2), 2)
```

True



# Testing Exceptions with unittest

```
def even_only(n: int) -> None:  
    """ raise ValueError if n is an odd number """  
    if n % 2 != 0:  
        raise ValueError(str(n) + "is an odd number")  
  
class EvenOnlyTest(unittest.TestCase):  
    def test_even_only(self):  
        with self.assertRaises(ValueError):  
            even_only(3)
```

Calling even\_only(3)  
should raise ValueError



# Testing Exceptions with unittest

```
class Triangle:  
    def __init__(self, s1: float, s2:float, s3: float) -> None:  
        if s1 <= 0 or s2 <= 0 or s3 <= 0:  
            raise ValueError("side <= 0")  
        else:  
            self.a: float = s1  
            self.b: float = s2  
            self.c: float = s3
```

Testing all sides  $> 0$  in `__init__()` means that you can't create an invalid triangle

```
class TriangleTest(unittest.TestCase):  
    def test_valid_side(self):  
        """ verify side is a number > 0 """  
        with self.assertRaises(ValueError):  
            Triangle(-3, 3, 3)  
        with self.assertRaises(ValueError):  
            Triangle(3, -3, 3)  
        with self.assertRaises(ValueError):  
            Triangle(3, 3, -3)
```

Creating an instance of Triangle with any side  $\leq 0$  should raise ValueError



# Writing test cases

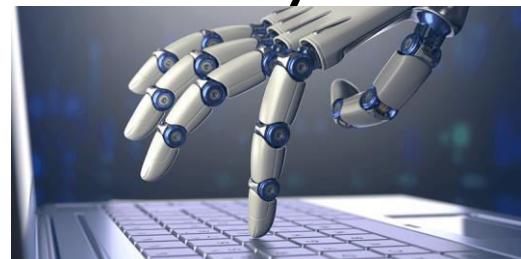
Not everyone uses Test Driven Development when writing code or perhaps the tests are incomplete  
(or homework assignments require you to write tests ;-)

You may be asked to add to the test suite of code you didn't write

Good testers are invaluable

Good test cases are as valuable as good code

Good code without test cases is hard to trust



# What can go wrong?

Logic/code errors

Misunderstood/unclear requirements

Unexpected parameter values

Boundary conditions

Incomplete test suites

Others...





# Test boundary conditions: abs(n)

## Boundary conditions: negative, 0, positive

```
import unittest
```

```
def buggy_abs(n: float) -> float:  
    """ This implementation of absolute value has some problems.  
        Hopefully our tests will catch them  
    """  
  
    if n > 0:  
        return n  
    elif n < 0:  
        return -n
```

```
class BuggyABSTest(unittest.TestCase):  
    def test_buggy_abs(self) -> None:  
        self.assertEqual(buggy_abs(3), 3)  
        self.assertEqual(buggy_abs(-3), 3)  
        self.assertEqual(buggy_abs(0), 0)
```

```
if __name__ == '__main__':  
    test_buggy_abs (__main__.BuggyABSTest) ... ok
```

---

Ran 1 test in 0.000s

OK

Python is trying to warn us about a problem

Tested positive and negative numbers  
→ All tests passed! But wait:  
What's buggy\_abs(0)? None Why?  
We need to add a test for buggy\_abs(0) == 0



# Test boundary conditions: abs(n)

## Boundary conditions: negative, 0, positive

```
import unittest

def buggy_abs(n: float) -> float:
    """ This implementation of absolute value has some problems.
        Hopefully our tests will catch them
    """
    if n > 0:
        return n
    elif n < 0:
        return -n

class BuggyABSTest(unittest.TestCase):
    def test_buggy_abs(self) -> None:
        self.assertEqual(buggy_abs(3), 3)
        self.assertEqual(buggy_abs(-3), 3)
        self.assertEqual(buggy_abs(4), buggy_abs(-4))
        self.assertEqual(buggy_abs(0), 0)

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)

test_buggy_abs (__main__.BuggyABSTest) ... FAIL
```

Added missing  
test case

We added the test for `buggy_abs(0)` and our test suite  
correctly points out that we have more work to do.  
We'll talk about that in the next section.

```
=====
FAIL: test_buggy_abs (__main__.BuggyABSTest)
-----
Traceback (most recent call last):
  File "/Users/jrr/Documents/Stevens/810/examples/03-buggyABS-unittest.py", line 17, in test_buggy_abs
    self.assertEqual(buggy_abs(0), 0)
AssertionError: None != 0
```

Ran 1 test in 0.001s  
FAILED (failures=1)

Test shows we  
have a problem

# Testing Strategies

**Test each function**

**Test expected case**

**Test boundary conditions** and “around the edges”

What might have been missed?

**Include both positive and negative tests**

**Positive:** valid input produces valid output

**Negative:** invalid input is handled gracefully

Beware that tests can have bugs too!

Missing important tests

Bugs in the test case





# Are we done yet??

```
import unittest

def less_than(a: float, b: float) -> bool:
    """ return True if a < b else False """
    return True

class LessThanTest(unittest.TestCase):
    def test_less_than(self) -> None:
        self.assertTrue(less_than(2, 3)) # enough testing, right?

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)
```

2 < 3 should be True  
What's the problem???

What other tests should  
we include?

What tests do we need  
for a <= b?



# Source file structure

# Organize your code into two files

- I. Source code for classes and functionality
  - II. Automated test cases

HW03\_Jim\_Rowland.py

```
""" Implementation of absolute value function """

def buggy_abs(n: float) -> float:
    """ This implementation of absolute value has some problems.
        Hopefully our tests will catch them
    """
    if n >= 0:
        return n
    elif n < 0:
        return -n
    else:
        raise ValueError("Unexpected value in buggy_abs")

if __name__ == "__main__":
    print("Hello from buggy abs")
```

No print statements here  
to encourage others to  
reuse our code

No print statements here  
to encourage others to  
reuse our code

HW03 Test Jim Rowland.py

```
""" automated tests for buggy_abs """
import unittest
from HW03_Jim_Rowland import buggy_abs

class BuggyABSTest(unittest.TestCase):
    """ test buggy_abs """
    def test_buggy_abs(self) -> None:
        """ buggy_abs test cases """
        self.assertEqual(buggy_abs(3), 3)
        self.assertEqual(buggy_abs(-3), 3)
        self.assertEqual(buggy_abs(4), buggy_abs(-4))
        self.assertEqual(buggy_abs(0), 0)

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)
```

And now  
for something  
completely different...



# Debugging

We've explored how we can use test cases to identify bugs in new and existing functions

Once we know there's a bug, how do we fix it?

What tools and techniques can we use?

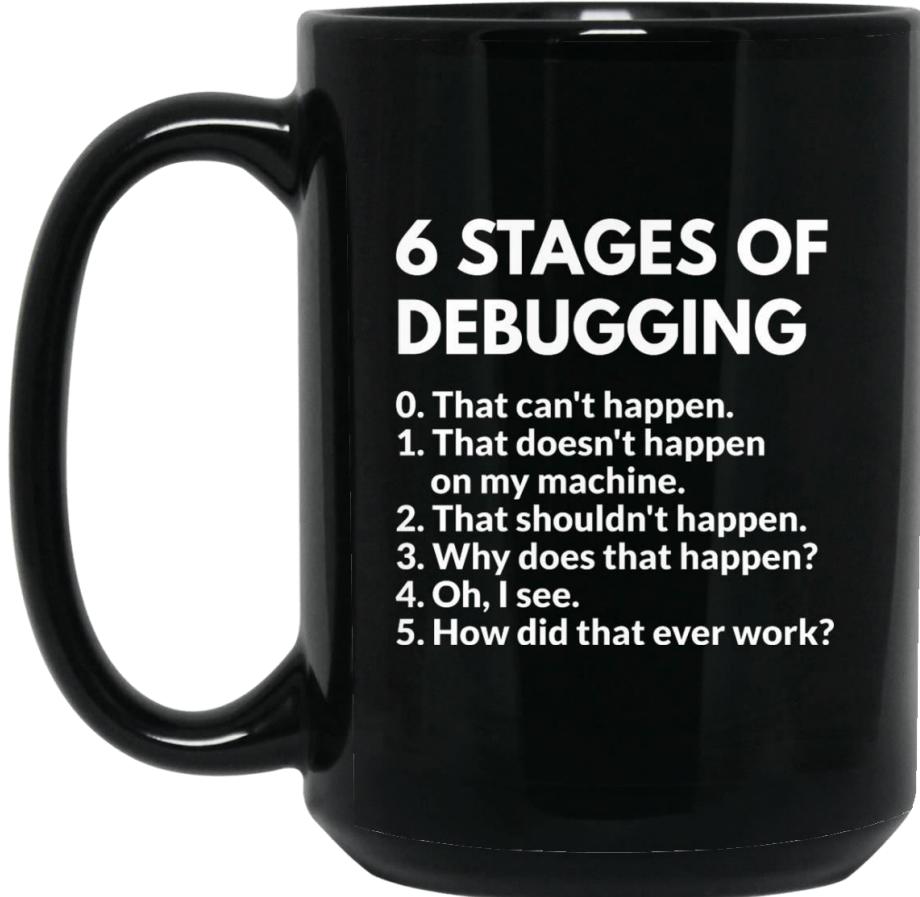
Debugging is hard, but there are tools to help

Trying to find problems without debugging tools is like trying to find a needle in a haystack





# Just for fun



# Debugging is hard

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?”

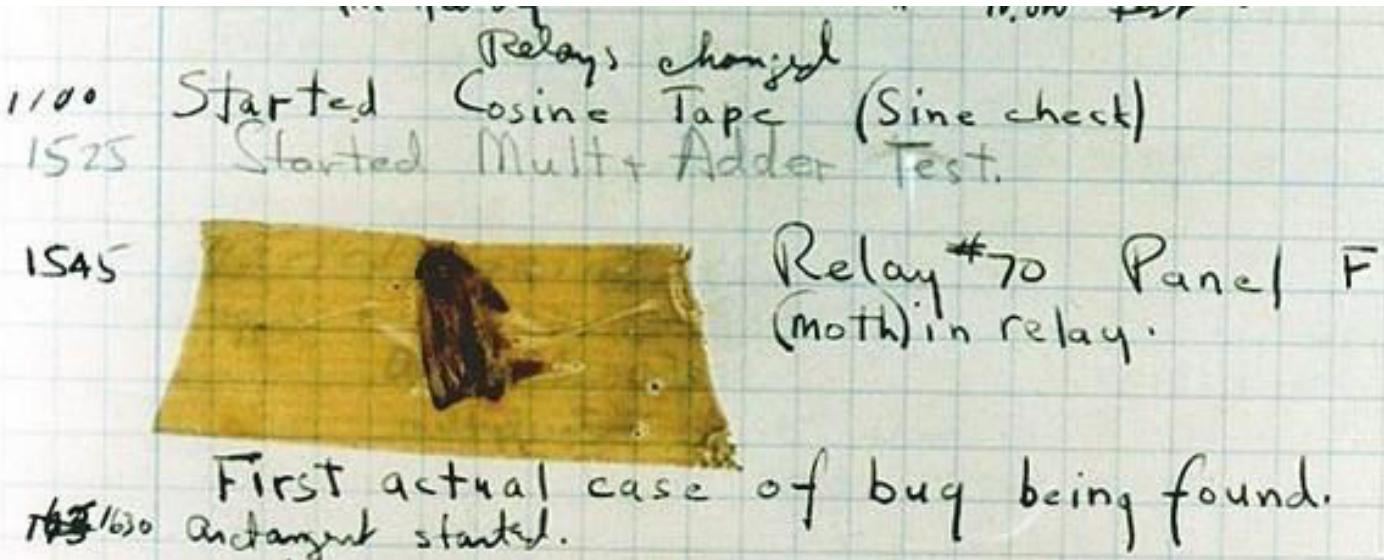
-- Brian W. Kernighan



# The first (but not the last) bug

The first bug was found by Admiral Grace Hopper at Harvard University in 1947 – a moth that impacted the hardware

<http://www.computerhistory.org/tdih/September/9/>





# Test boundary conditions: abs(n)

## Boundary conditions: negative, 0, positive

```
import unittest

def buggy_abs(n: float) -> float:
    """ This implementation of absolute value has some problems.
        Hopefully our tests will catch them
    """
    if n > 0:
        return n
    elif n < 0:
        return -n

class BuggyABSTest(unittest.TestCase):
    def test_buggy_abs(self) -> None:
        self.assertEqual(buggy_abs(3), 3)
        self.assertEqual(buggy_abs(-3), 3)
        self.assertEqual(buggy_abs(4), buggy_abs(-4))
        self.assertEqual(buggy_abs(0), 0)

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)

test_buggy_abs (__main__.BuggyABSTest) ... FAIL
```

```
=====
FAIL: test_buggy_abs (__main__.BuggyABSTest)
-----
Traceback (most recent call last):
  File "/Users/jrr/Documents/Stevens/810/examples/03-buggyABS-unittest.py", line 17, in test_buggy_abs
    self.assertEqual(buggy_abs(0), 0)
AssertionError: None != 0
```

```
Ran 1 test in 0.001s
```

```
FAILED (failures=1)
```

Something is wrong with the logic for 0.  
How do we find it and fix it?

# 3 Steps to debugging

Step 1: Recognize that there is a problem (testing is one solution)

Step 2: Find the cause of the problem

Step 3: Fix the problem without breaking anything else

Automated testing helps to verify that the fix didn't break anything

All three can be challenging! That's why good software engineers are in such high demand (and earn such high salaries ;-)





# Debugging techniques

Once you know there is a bug, how do you find it?

Approach	Advantages	Disadvantages
Read the code	Code reviews are the most effective way to find problems	Difficult and time consuming, especially with unfamiliar, or large code. Most effective with highly experienced readers
Print statements	Simplicity – anyone can do it.	May be difficult to add enough print statements at the right place to be efficient. Must remove print statements
Static code analyzers	No code changes needed. Helps to find “silly”, but subtle errors	None
Debuggers	Fast, efficient, thorough	Small learning curve, but it's a skill that will save countless hours

# Debugging: Read the code

## Test suite identified a problem

```
import unittest

def buggy_abs(n: float) -> float:
    """ This implementation of absolute value has some problems.
        Hopefully our tests will catch them
    """
    if n > 0:
        return n
    elif n < 0:
        return -n

class BuggyABSTest(unittest.TestCase):
    def test_buggy_abs(self) -> None:
        self.assertEqual(buggy_abs(3), 3)
        self.assertEqual(buggy_abs(-3), 3)
        self.assertEqual(buggy_abs(4), buggy_abs(-4))
        self.assertEqual(buggy_abs(0), 0)

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)
```

```
test_buggy_abs (__main__.BuggyABSTest) ... FAIL
=====
FAIL: test_buggy_abs (__main__.BuggyABSTest)
-----
Traceback (most recent call last):
  File "/Users/jrr/Documents/Stevens/810/examples/03-buggyABS-unittest.py", line 17, in test_buggy_abs
    self.assertEqual(buggy_abs(0), 0)
AssertionError: None != 0

Ran 1 test in 0.001s
FAILED (failures=1)
```

Testing reports bug when  $n == 0$ . Read the code to see if we can find the problem.  
AH HA!!! The logic handles  $< 0$  and  $> 0$  but not  $0$ .  
Now we know the problem. Next, how to solve it?





# Debugging: Add print statements

## Test suite identified a problem

```
def buggy_abs(n: float) -> float:  
    """ This implementation of absolute value has some problems.  
        Hopefully our tests will catch them  
    ....  
  
    if n > 0:  
        print(f"Debug: n > 0 - return {n}")  
        return n  
    elif n < 0:  
        print(f"Debug: n < 0 - return {n * -1}")  
        return -n  
    print("Debug: shouldn't be here")  
  
class BuggyABSTest(unittest.TestCase):  
    def test_buggy_abs(self) -> None:  
        self.assertEqual(buggy_abs(3), 3)  
        self.assertEqual(buggy_abs(-3), 3)  
        self.assertEqual(buggy_abs(4), buggy_abs(-4))  
        self.assertEqual(buggy_abs(0), 0)  
  
if __name__ == '__main__':  
    unittest.main(exit=False, verbosity=2)  
  
test_buggy_abs (__main__.BuggyABSTest) ... Debug: n > 0 - return 3  
Debug: n < 0 - return 3  
Debug: n > 0 - return 4  
Debug: n < 0 - return 4  
Debug: shouldn't be here  
FAIL
```

Testing reports a bug when  $n == 0$ .  
Added print statements to help to understand what's happening.

AH HA!!! The logic handles  $< 0$  and  $> 0$  but we fall off the end of the function with 0.



Now we know the problem. Next, how to solve it?

Oh yeah, don't forget to remove those print statements when we're done!

# Fixing the problem

The test suite identified a problem

We read the code and used print statements to understand that 0 falls through the if/else statement and the function is returning None

Solution: change the if statement to handle the 0 case



# Verify our change

Did our change fix the problem? Did we break anything else?

```
import unittest

def buggy_abs(n: float) -> float:
    """ This implementation of absolute value has some problems.
        Hopefully our tests will catch them
    """
    if n >= 0: _____
        print(f"Debug: n >= 0 - return {n}")
        return n
    elif n < 0:
        print(f"Debug: n < 0 - return {n * -1}")
        return -n
    print("Debug: shouldn't be here")

class BuggyABSTest(unittest.TestCase):
    def test_buggy_abs(self) -> None:
        self.assertEqual(buggy_abs(3), 3)
        self.assertEqual(buggy_abs(-3), 3)
        self.assertEqual(buggy_abs(4), buggy_abs(-4))
        self.assertEqual(buggy_abs(0), 0)

if __name__ == '__main__':
    unittest.main(exit=False, verbosity=2)

test_buggy_abs (__main__.BuggyABSTest) ... Debug: n >= 0 - return 3
Debug: n < 0 - return 3
Debug: n >= 0 - return 4
Debug: n < 0 - return 4
Debug: n >= 0 - return 0
ok
```

Changed > to >=

All tests pass after our fix!  
Oh yeah, don't forget to remove those print statements!





# Debugging tools: Static code analysis

Tools read your Python code and report “violations”

Syntax errors

Uninitialized variables

Undefined functions

Coding style violations

Measures coding standards compliance and assigns a score

We'll focus first on Pylint integration with VS Code



# Debugging tools: Debuggers

Set breakpoints in the code to pause execution and explore

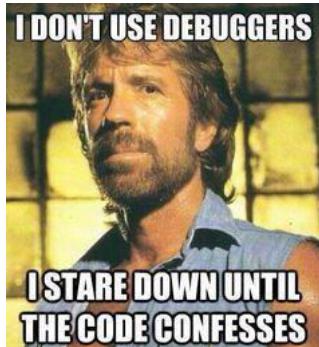
Step through code line by line

Step **into** function calls/Step **out of** function calls

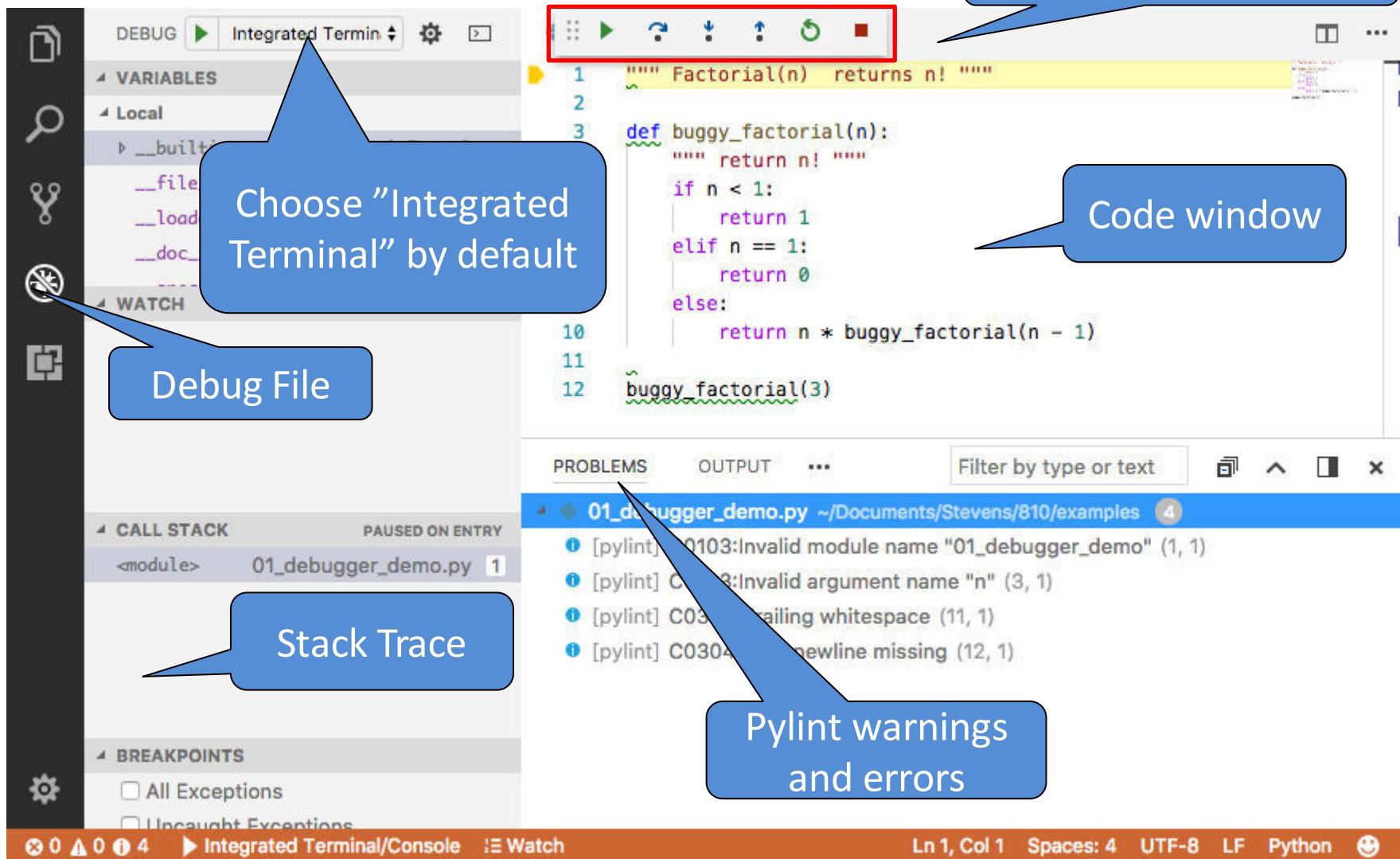
Watch variables as they change values during execution

Evaluate arbitrary expressions

Investigate the call stack, i.e. which functions called what



# Debugging tools: VS Code



The screenshot shows the VS Code interface with several features highlighted:

- Integrated Terminal**: A dropdown menu item in the top-left corner.
- Debugger Commands**: A speech bubble pointing to the toolbar above the code editor.
- Code window**: A speech bubble pointing to the main code editor area.
- Pylint warnings and errors**: A speech bubble pointing to the Problems panel at the bottom.
- Stack Trace**: A speech bubble pointing to the Call Stack panel on the left.
- Debug File**: A speech bubble pointing to the Debug sidebar on the left.
- Choose "Integrated Terminal" by default**: A speech bubble pointing to the Integrated Terminal dropdown menu.

```
1 """ Factorial(n)  returns n! """
2
3 def buggy_factorial(n):
4     """ return n! """
5     if n < 1:
6         return 1
7     elif n == 1:
8         return 0
9     else:
10        return n * buggy_factorial(n - 1)
11
12 buggy_factorial(3)
```

PROBLEMS OUTPUT ... Filter by type or text

- [pylint] C0103:Invalid module name "01\_debugger\_demo" (1, 1)
- [pylint] C0103:Invalid argument name "n" (3, 1)
- [pylint] C0301:Trailing whitespace (11, 1)
- [pylint] C0304:Newline missing (12, 1)

CALL STACK PAUSED ON ENTRY

01\_debugger\_demo.py ~/Documents/Stevens/810/examples 4

01\_debugger\_demo.py 1

BREAKPOINTS

All Exceptions

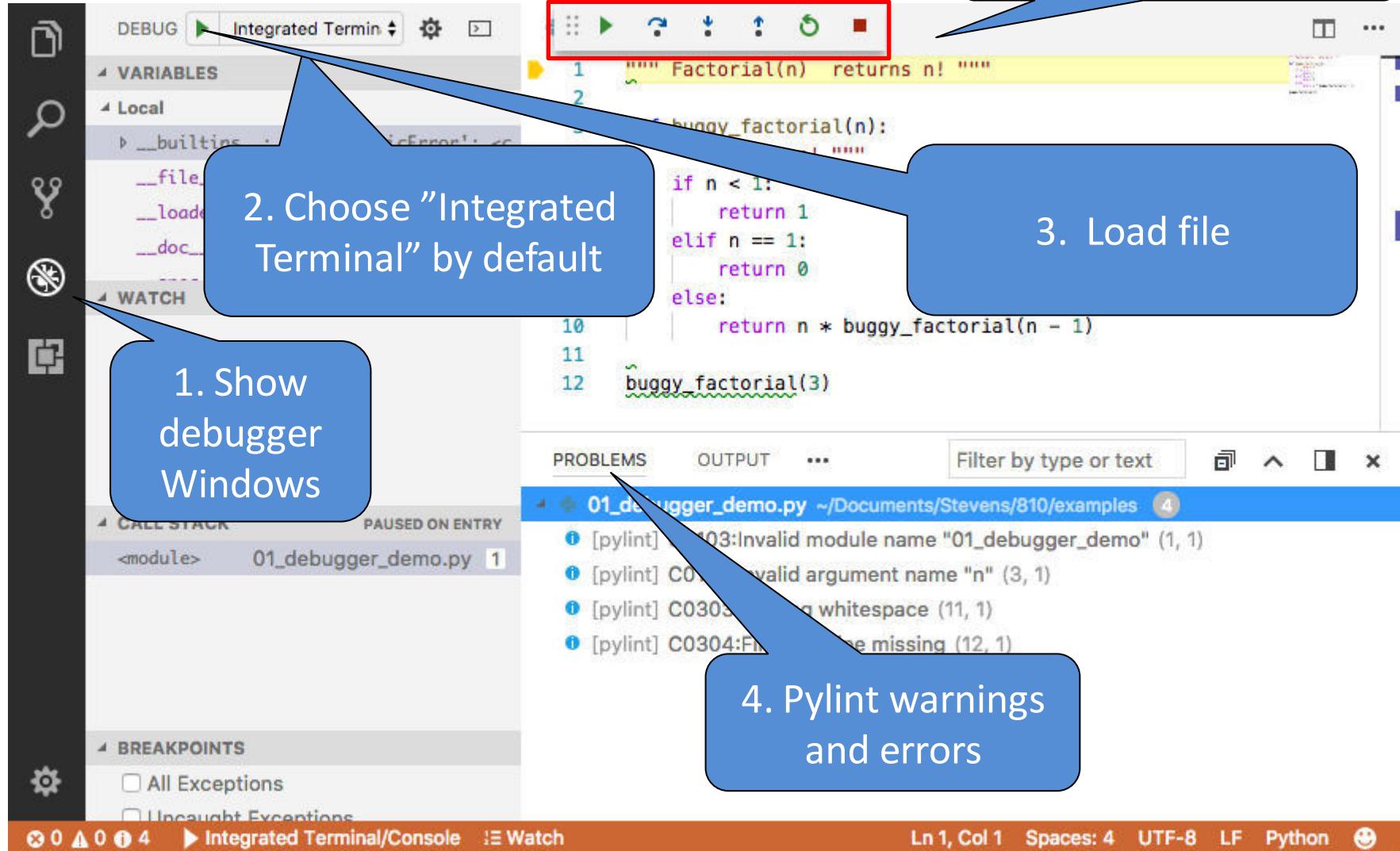
Uncaught Exceptions

Integrated Terminal/Console Watch

Ln 1, Col 1 Spaces: 4 UTF-8 LF Python 😊

# Debugging tools: VS Code

Debugger Commands



1. Show debugger Windows

2. Choose "Integrated Terminal" by default

3. Load file

4. Pylint warnings and errors

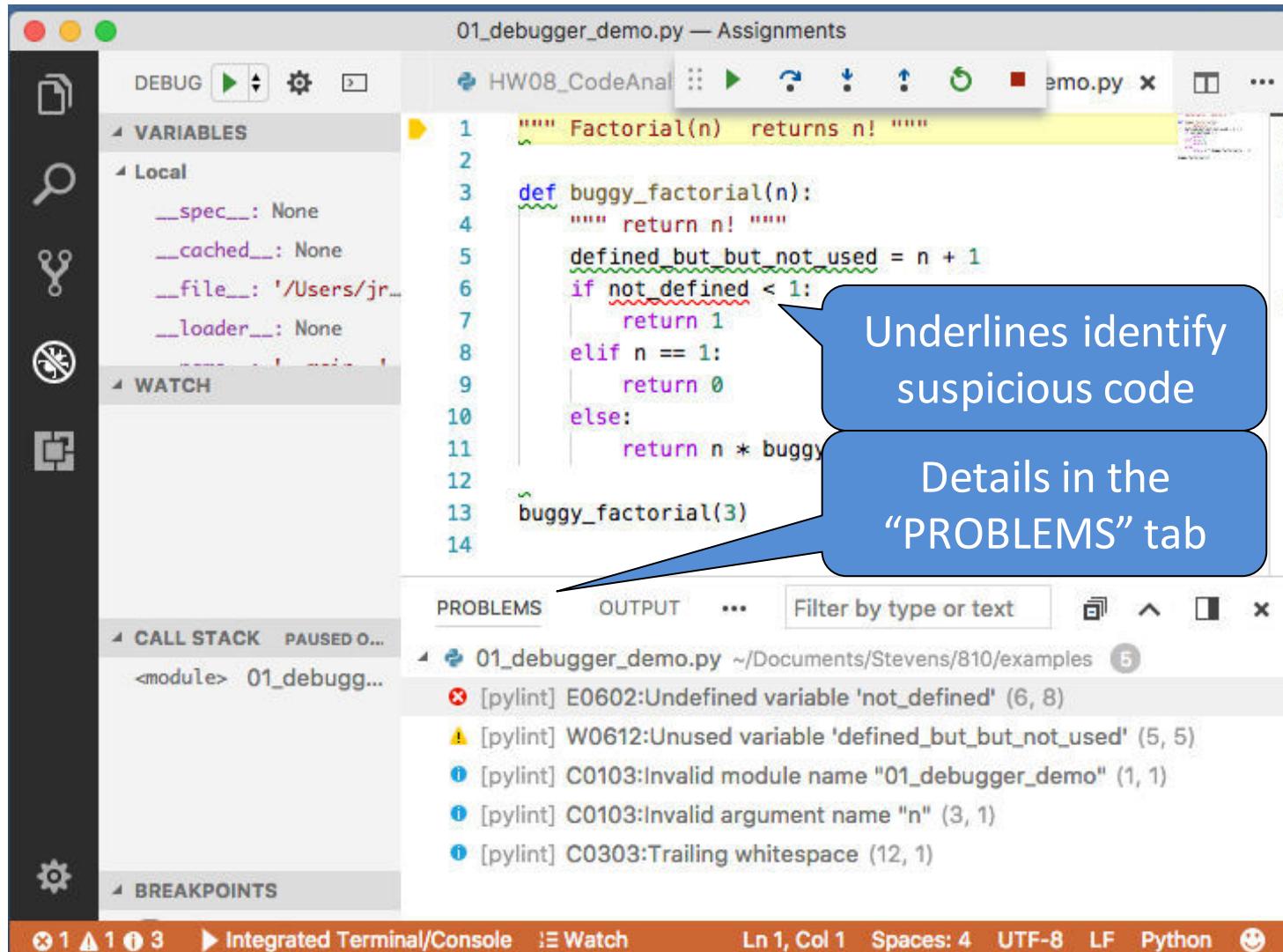
The screenshot shows the VS Code interface with the following details:

- Top Bar:** DEBUG button is highlighted. The integrated terminal dropdown shows "Integrated Terminal".
- Left Sidebar:** Shows icons for Files, Search, Yarn, and Terminal. The "VARIABLES" section is expanded, showing Local variables like \_\_builtins\_\_, \_\_file\_\_, \_\_loader\_\_, \_\_doc\_\_, etc.
- Code Editor:** Displays Python code for a factorial function with a bug. The code is:

```
1 """ Factorial(n) returns n! """
2
3 def buggy_factorial(n):
4     if n < 1:
5         return 1
6     elif n == 1:
7         return 0
8     else:
9         return n * buggy_factorial(n - 1)
10
11
12 buggy_factorial(3)
```
- Bottom Status Bar:** Shows 00 0 0 4 Integrated Terminal/Console Watch. Status bar: Ln 1, Col 1 Spaces: 4 UTF-8 LF Python ☀️
- Bottom Right:** A blue callout box labeled "Debugger Commands" points to the top bar's DEBUG button.
- Bottom Left:** A blue callout box labeled "1. Show debugger Windows" points to the sidebar icon.
- Middle Left:** A blue callout box labeled "2. Choose 'Integrated Terminal' by default" points to the integrated terminal dropdown.
- Middle Right:** A blue callout box labeled "3. Load file" points to the code editor area.
- Bottom Right:** A blue callout box labeled "4. Pylint warnings and errors" points to the Problems panel, which lists four pylint errors:

Severity	Message	Line	Column
Info	[pylint] E003:Invalid module name "01_debugger_demo"	(1, 1)	
Info	[pylint] C0201:invalid argument name "n"	(3, 1)	
Info	[pylint] C0303:using whitespace	(11, 1)	
Info	[pylint] C0304:File missing	(12, 1)	

# Debugging tools: Code warnings

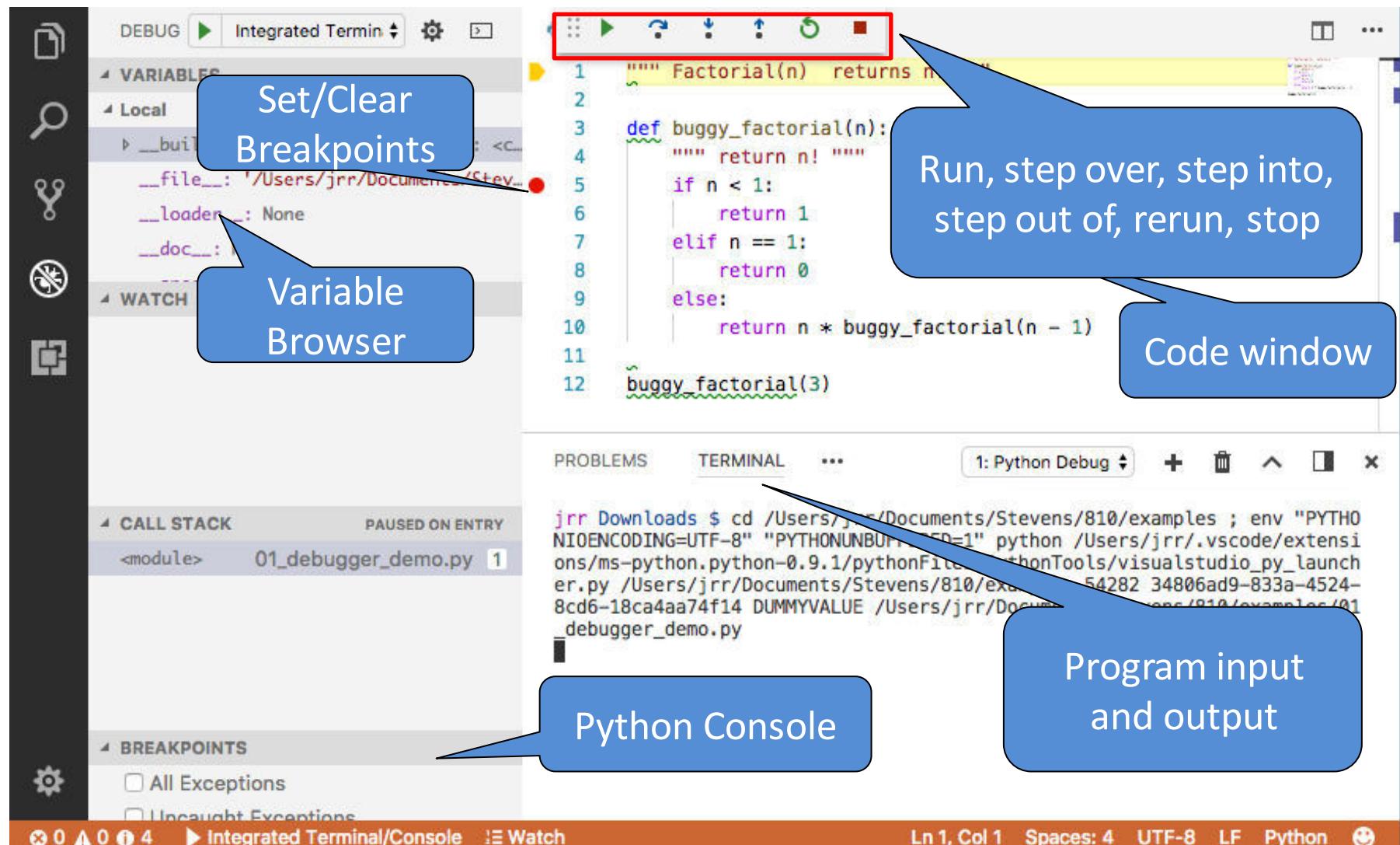


The screenshot shows a Python debugger interface with the following details:

- Code Editor:** The file `01_debugger_demo.py` is open, showing a buggy factorial function:

```
1 """ Factorial(n)  returns n! """
2
3 def buggy_factorial(n):
4     """ return n! """
5     defined_but_but_not_used = n + 1
6     if not_defined < 1:
7         return 1
8     elif n == 1:
9         return 0
10    else:
11        return n * buggy
12
13 buggy_factorial(3)
14
```
- VARIABLES Panel:** Shows local variables: `__spec__`, `__cached__`, `__file__`, `__loader__`.
- PROBLEMS Tab:** Displays analysis results from `HW08_CodeAnal`:
  - 01\_debugger\_demo.py** (~/Documents/Stevens/810/examples)
  - Issues:**
    - [pylint] E0602:Undefined variable 'not\_defined' (6, 8)
    - [pylint] W0612:Unused variable 'defined\_but\_but\_not\_used' (5, 5)
    - [pylint] C0103:Invalid module name "01\_debugger\_demo" (1, 1)
    - [pylint] C0103:Invalid argument name "n" (3, 1)
    - [pylint] C0303:Trailing whitespace (12, 1)
- Annotations:** Two callout boxes highlight features:
  - A blue box points to the underlined code in the editor: "Underlines identify suspicious code".
  - A blue box points to the PROBLEMS tab: "Details in the ‘PROBLEMS’ tab".

# Debugging tools: VS Code





# Debugging Commands

## Set Breakpoint

Double click left of the line number to set breakpoint

## Debug File

Run the file and stop at first breakpoint

## Run Current Line

Execute current line

## Step into function

Execute current line and step into function calls

The screenshot shows the VS Code interface with the Python extension loaded. The top bar has 'Run File' and 'Run Current Line' buttons. The main area shows a Python script with breakpoints set at line 5 and line 12. The 'Variables' sidebar shows local arguments: n=3. The 'Breakpoints' sidebar lists two entries: '01\_debugger\_demo.py' and '01\_debugger\_demo.py'. The terminal at the bottom shows the command to run the debugger.

```
jrr Downloads $ cd /Users/jrr/Documents/Stevens/810/exa  
=1" python /Users/jrr/.vscode/extensions/ms-python.pyth  
her.py /Users/jrr/Documents/Stevens/810/examples 57449  
/jrr/Documents/Stevens/810/examples/01_debugger_demo.py
```

# Debugging Commands

**Step out of function**

Continue execution and return to calling function

**Debug File**

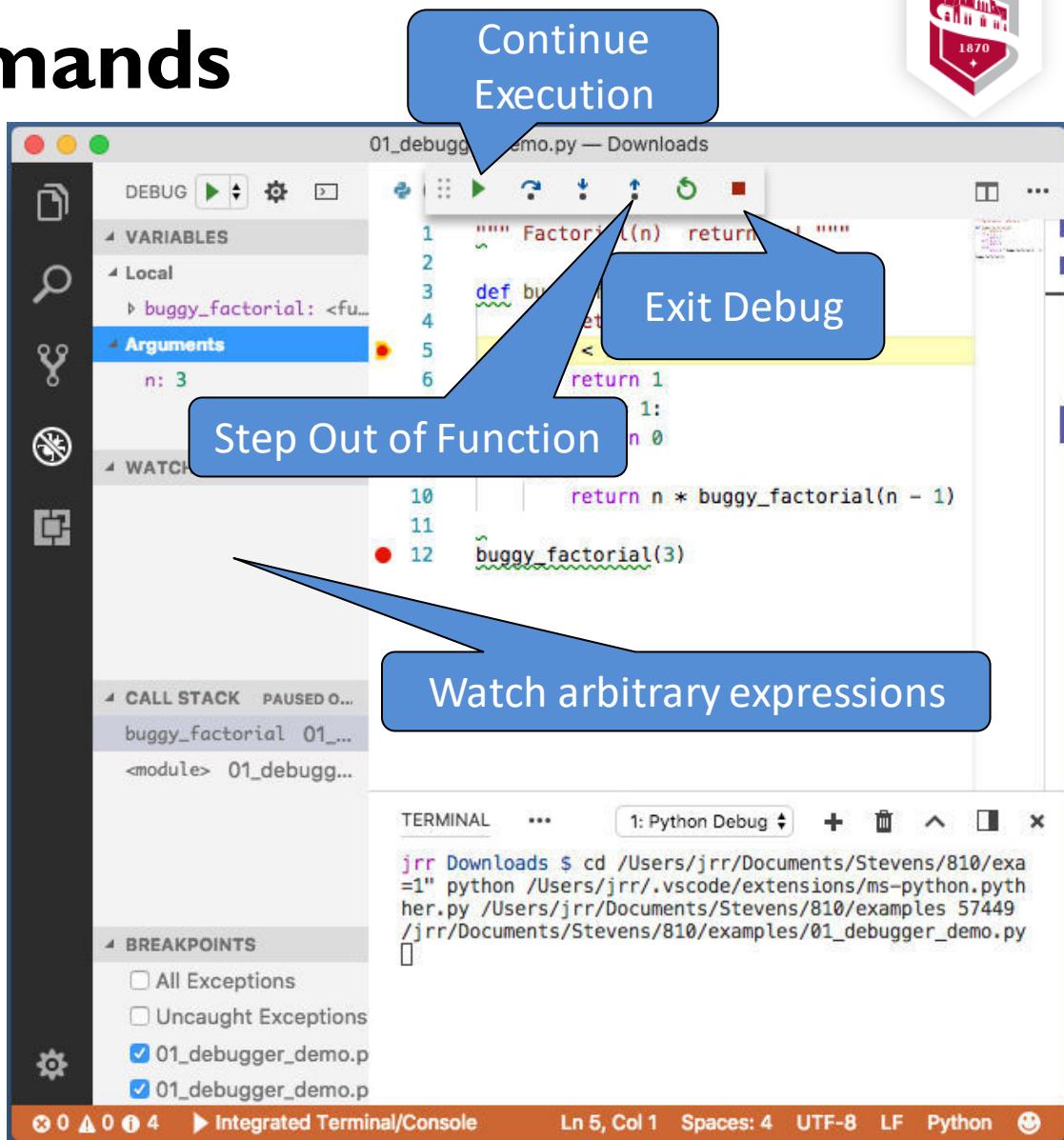
Run the file and stop at first breakpoint

**Continue Execution**

Continue execution until next breakpoint

**Exit Debug**

Stop debugging



01\_debugger\_demo.py — Downloads

DEBUG ➜ ⚙️ ⚙️ ⚙️

VARIABLES

Local

Arguments

n: 3

WATCH

CALL STACK PAUSED ON

buggy\_factorial 01...

<module> 01\_debugger...

BREAKPOINTS

All Exceptions

Uncaught Exceptions

01\_debugger\_demo.p

01\_debugger\_demo.p

0 0 ▲ 0 0 4 Integrated Terminal/Console

1: """ Factorial(n) returns n! """
2:
3: def buggy\_factorial(n):
4: if n == 0:
5: return 1
6: else:
7: return n \* buggy\_factorial(n - 1)
8:
9: print(buggy\_factorial(3))
10:
11:
12: buggy\_factorial(3)

TERMINAL 1: Python Debug

```
jrr Downloads $ cd /Users/jrr/Documents/Stevens/810/exa
=1" python /Users/jrr/.vscode/extensions/ms-python.pyth
her.py /Users/jrr/Documents/Stevens/810/examples 57449
/jrr/Documents/Stevens/810/examples/01_debugger_demo.py
```

Ln 5, Col 1 Spaces: 4 UTF-8 LF Python

Continue Execution

Exit Debug

Step Out of Function

Watch arbitrary expressions



# Debugging: Typical session

1. Set a breakpoint or two

Where??? `main()` is one choice

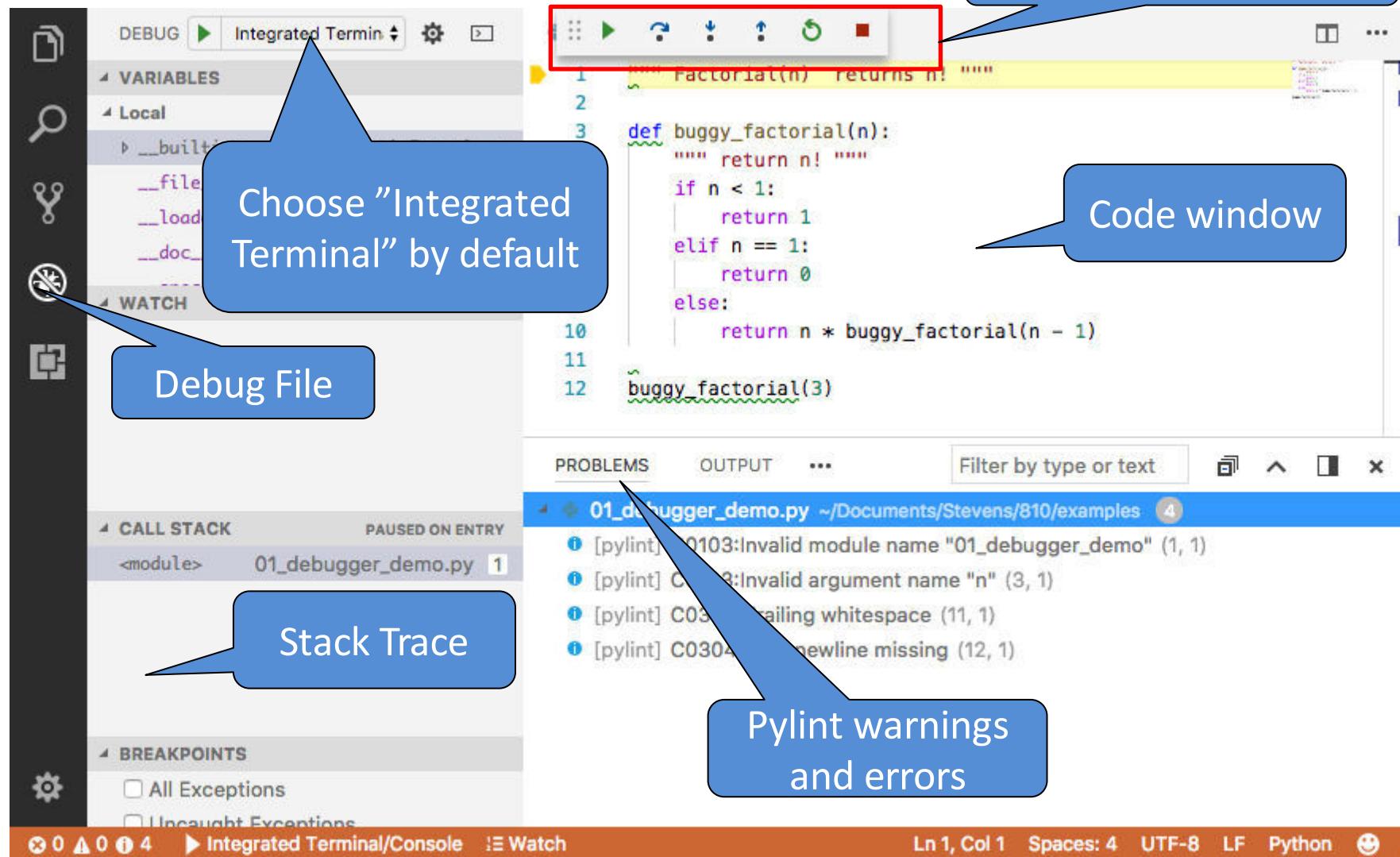
2. Debug File (run debugger)
3. Step across/into/out of function calls

4. Evaluate variables

Variable Explorer

5. Stop and think about what's happening and why

# Live Debugger Demo: VS Code



The screenshot shows the VS Code interface with several features highlighted:

- Integrated Terminal**: A blue callout points to the terminal icon in the top bar.
- VARIABLES**: A blue callout points to the variables sidebar.
- Local**: A blue callout points to the local variable section in the variables sidebar.
- WATCH**: A blue callout points to the watch sidebar.
- CALL STACK**: A blue callout points to the call stack sidebar.
- PAUSED ON ENTRY**: A blue callout points to the status message "PAUSED ON ENTRY".
- BREAKPOINTS**: A blue callout points to the breakpoints sidebar.
- All Exceptions**: A checkbox in the breakpoints sidebar.
- Uncaught Exceptions**: A checkbox in the breakpoints sidebar.
- Integrated Terminal/Console**: A blue callout points to the integrated terminal tab.
- Watch**: A blue callout points to the watch tab.
- DEBUG**: A blue callout points to the debug icon in the top bar.
- Code window**: A blue callout points to the main code editor area.
- Debugger Commands**: A blue callout points to the toolbar above the code editor.
- Stack Trace**: A blue callout points to the stack trace sidebar.
- Pylint warnings and errors**: A blue callout points to the problems sidebar.
- PROBLEMS**: A blue callout points to the problems tab.
- OUTPUT**: A blue callout points to the output tab.
- Filter by type or text**: A text input field in the problems sidebar.
- 01\_debugger\_demo.py**: The active file in the code editor.
- Line numbers**: Line 1, Col 1, Spaces: 4, UTF-8, LF, Python, and a smiley face icon at the bottom.

```
def buggy_factorial(n):
    """ return n! """
    if n < 1:
        return 1
    elif n == 1:
        return 0
    else:
        return n * buggy_factorial(n - 1)

buggy_factorial(3)
```

# Occam's Razor for debugging

## CORE PRINCIPLES IN RESEARCH



### OCCAM'S RAZOR

"WHEN FACED WITH TWO POSSIBLE EXPLANATIONS, THE SIMPLER OF THE TWO IS THE ONE MOST LIKELY TO BE TRUE."



### OCCAM'S PROFESSOR

"WHEN FACED WITH TWO POSSIBLE WAYS OF DOING SOMETHING, THE MORE COMPLICATED ONE IS THE ONE YOUR PROFESSOR WILL MOST LIKELY ASK YOU TO DO."

[WWW.PHDCOMICS.COM](http://WWW.PHDCOMICS.COM)

JOE GAGNE CHAM © 2009



# Debugging Heuristics and Strategies

Divide and Conquer/Binary search

Minimize and isolate the code and data being debugged

New code is more likely to have bugs than older code

Try to get back to a working state

Boundary cases are always suspect

Beware of integration points with other modules and code authors

Check the easy things first, e.g. silly mistakes

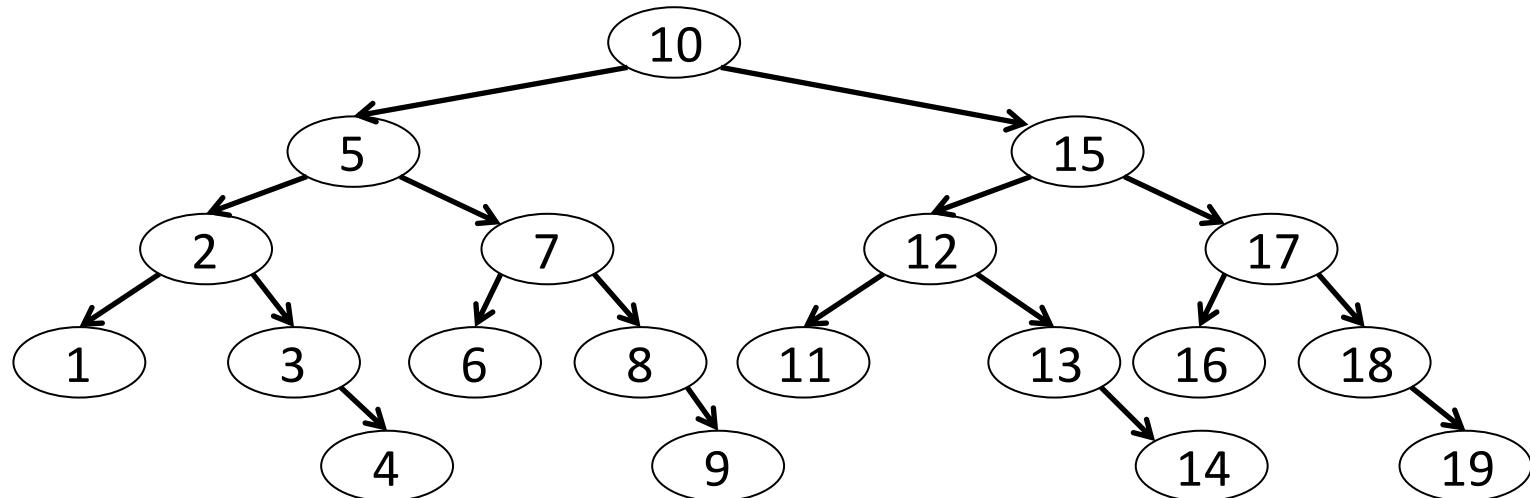
Learn from your mistakes so you don't make them again

# Divide and Conquer/Binary Search

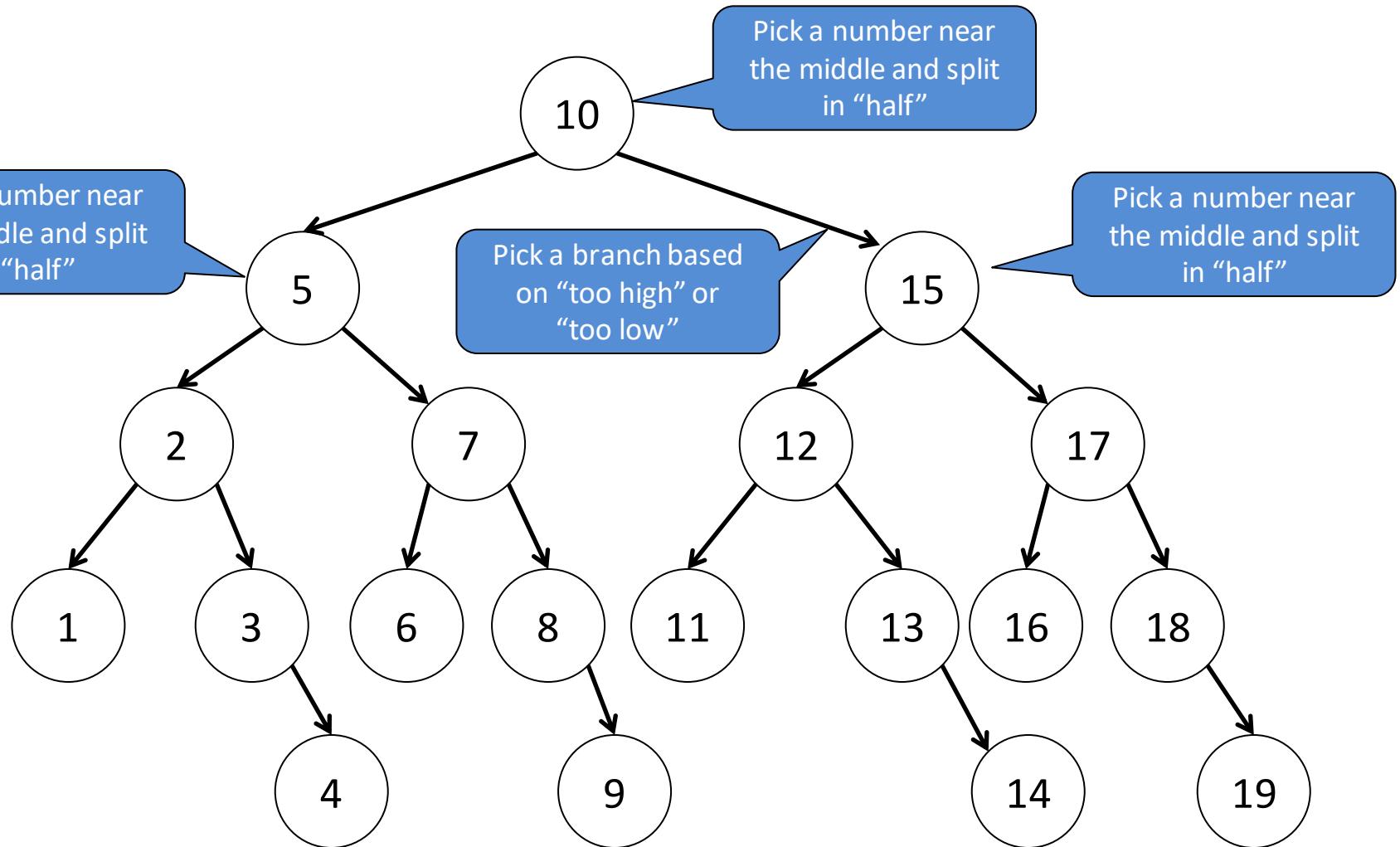
Goal is to isolate the problem to minimize the places the bug can be hiding

Remember, the bug may be in the code or the data

Cut the search space in half and try to isolate the problem to one half or the other



# Guess a number between 1 and 20



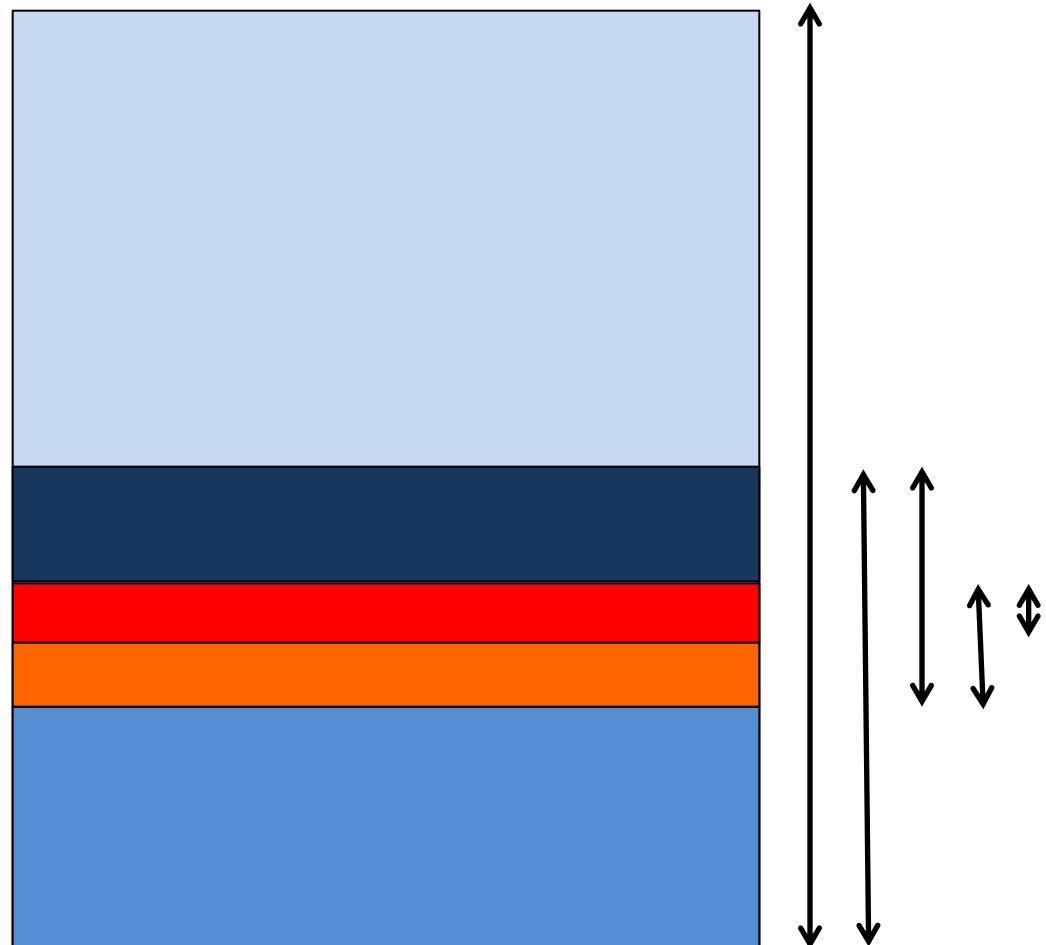
# Cut the search in half, test, repeat...

Isolate the problem to  
half of the problem  
space

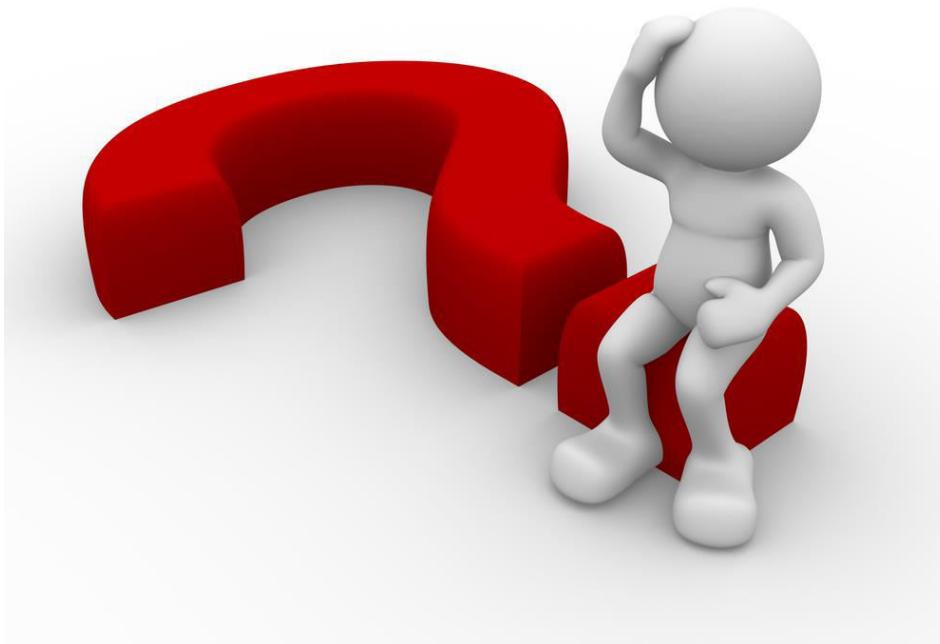
Reduce the code

Reduce the data

Reduce the space we  
need to search



# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Iteration*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

And

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

# Today's topics

Sequences and Iterators

for loops

range()

while loops

Generators





# Sequences and iterators

Python supports several types of **sequences**

Both ordered and unordered collections of items

Sequences support **iterators** to iterate over the items

Examples of Sequences:

```
# Examples of |Python sequences
"We can iterate through characters in strings"

# iterate through items in a list
colors: List[str] = ['red', 'orange', 'yellow', 'blue']

# iterate through the keys, values, or keys/values in a dict
eng2sp: Dict[str, str] = {'one': 'uno', 'three': 'tres', 'two': 'dos'}

# iterate through items in a set
uniq: Set[str] = {'Python', '!Java', "Swift", "C++"}
```



# Iterating through the items in a list

Lists are ordered so the order is predictable

Software people like to start at 0, not 1

```
small_ints: List[int] = [0, 1, 2]
```

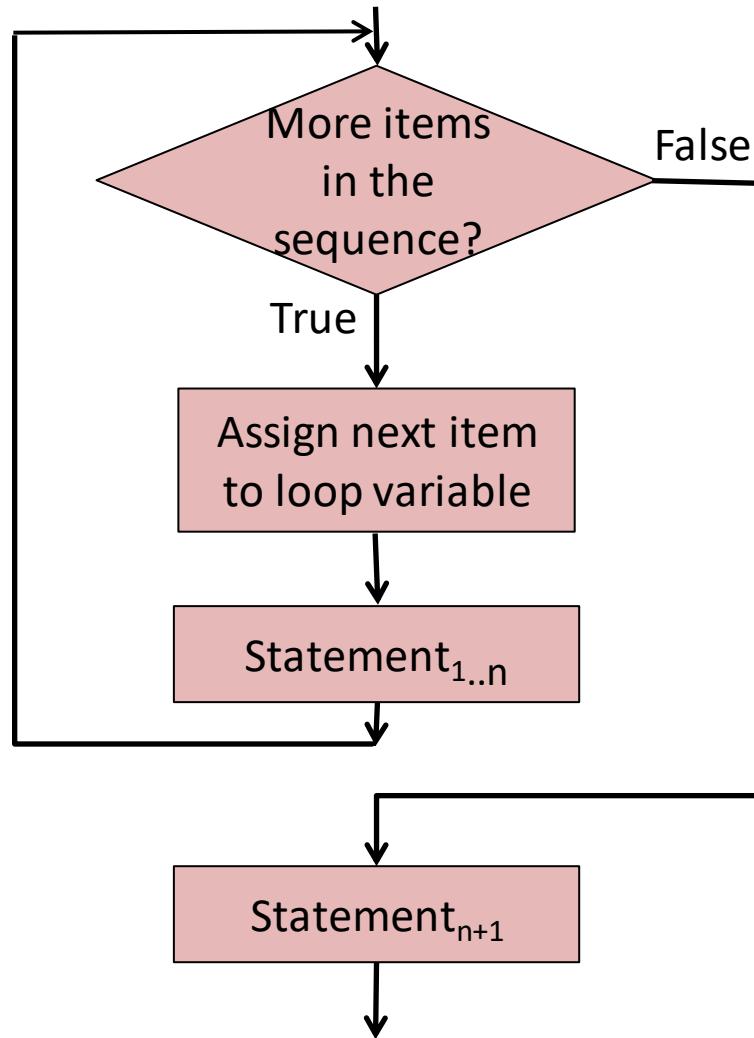
```
for i in small_ints:  
    print(i)
```

i gets the first, then each subsequent value of small\_ints

```
0  
1  
2
```

# Program flow: for loop

```
for item in sequence:  
    Statement1..n  
    Statementn+1
```





# Iterating through the items in a list

Let's calculate the sum of the values in `small_ints`

Iterate through the values from first to last

```
small_ints: List[int] = [0, 1, 2]
```

```
total: int = 0
for i in small_ints:
    print(i)
    total += i
print(f"total = {total}")
```

total will track the running total. Be sure to initialize it properly!

What if `small_ints` is empty?

```
0
1
2
total = 3
```



# Range of integers

```
small_ints: List[int] = [0, 1, 2]
```

Lists are very powerful, but what if we want the first 1000?

Other languages use counters

```
/* C/C++ for loops */  
for (i = 0; i < 1000; i++) {  
    cout << i;  
}
```



# range(): a new sequence type

for loops iterate over sequences

range() returns a sequence that supports iteration

```
for i in range(5):
    print(i)  # what is printed?
```

```
# sequence from 0 up to, but NOT including n
range(n)
```

```
# start at 'from' up to, NOT including 'to'
range(from, to)
```

```
# start at 'from' up to, NOT including 'to', increment by step
range(from, to, step)
```

From 0, up to, but **not** including *n*  
happens frequently in Python



# range(from, to, step)

range returns a **sequence**

```
range(5)
```

```
range(0, 5)
```

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
list(range(3, 5))
```

```
[3, 4]
```

```
list(range(3, 10, 3))
```

```
[3, 6, 9]
```

```
list(range(0, -5, -1))
```

```
[0, -1, -2, -3, -4]
```

You can see the values in  
the range by creating a list  
from the sequence



# Write a function...

... to calculate the sum of a range of integers

```
def sum_range(start: int, end: int) -> int:  
    """ Return the sum from start up to but not including end """  
    total: int = 0  
    for item in range(start, end):  
        total += item  
    return total  
  
assert sum_range(1, 1) == 0  
assert sum_range(1, 5) == 10  
assert sum_range(1, 2) == 1
```

0 ???  
Why not 1?



# Write a *Pythonic*\* function

Write a Pythonic function to calculate the sum of a range of integers from start, up to, but not including end

```
def py_sum_range(start: int, end: int) -> int:  
    """ Return the sum from start up to but not including end """  
    return sum(range(start, end))  
  
assert py_sum_range(1, 1) == 0  
assert py_sum_range(1, 5) == 10  
assert py_sum_range(1, 2) == 1
```

\*Writing Pythonic code means to think about how Python works and take advantage of built-in functionality, even if it's different from other programming languages



# For loops and strings

Strings are sequences too!

We can use **for** loops with strings (or any sequence)

```
for c in "Python":  
    print(c)
```

P  
y  
t  
h  
o  
n



# enumerate() and sequences

enumerate is a convenient way to assign an offset to an item in a sequence

```
for offset, c in enumerate("Python"):  
    print(offset, c)
```

```
0 P  
1 y  
2 t  
3 h  
4 o  
5 n
```

```
for offset, c in enumerate(['P', 'y', 't', 'h', 'o', 'n']):  
    print(offset, c)
```

```
0 P  
1 y  
2 t  
3 h  
4 o  
5 n
```



# Write a function: string\_length(str)

Use a `for` loop to calculate the length of a string

What happens if the string is empty?

```
def string_length(s: str) -> int:  
    cnt: int = 0  
    for c in s:  
        cnt += 1  
    return cnt  
  
assert string_length('Python') == 6  
assert string_length('') == 0  
assert string_length('Python') == len('Python')
```



# Write a function: `in_string(target, string)`

Use a `for` loop to look for a character in a string

```
def in_string(target: str, string: str) -> int:
    """ Check if target is in str.
        Return the offset from the beginning of the string
        or -1 if not found
    """
    for offset, c in enumerate(string):
        if c == target:
            return offset
    return -1 # if here then we didn't find the target

assert in_string('P', 'Python') == 0
assert in_string('t', 'Python') == 2
assert in_string('x', 'Python') == -1
assert in_string('x', '') == -1
```



# zip(\*sequences)

zip helps you to iterate through multiple sequences

```
seq1 = range(3) # [0, 1, 2]
seq2: List[str] = ['zero', 'one', 'two', 'three']
seq3: List[str] = ['zero', 'uno', 'dos', 'tres', 'quattro']

for digit, english, spanish in zip(seq1, seq2, seq3):
    print(digit, english, spanish)
```

```
0 zero zero
1 one uno
2 two dos
Zip(*sequences) sequence is exhausted
```

foo(\*args) takes any  
number of arguments



# Operations on sequences

Python provides built-in operations for all sequences

Operation	Description	Example
<code>len(s)</code>	Return the number of items in the sequence	<code>len('hello') == 5</code> <code>len(['a', 1, 3.4]) == 3</code>
<code>min(s)</code>	Return the min value	<code>min('hello') == 'e'</code> <code>min([1, 3.2, 7]) == 1</code>
<code>max(s)</code>	Return the max value	<code>max('hello') == 'o'</code> <code>max([1, 3.2, 7]) == 7</code>
<code>value in s</code>	Return True if value is in s else return False	'e' in 'hello' is True 3.2 in [1, 3.2, 7] is True
<code>value not in s</code>	Return True if s is not a member of s else return False	'x' not in 'hello' is True 2 not in [1, 3.2, 7] is True

How would you implement these operations with loops?



# More type hints

```
from typing import Sequence, Optional, Any, List
```

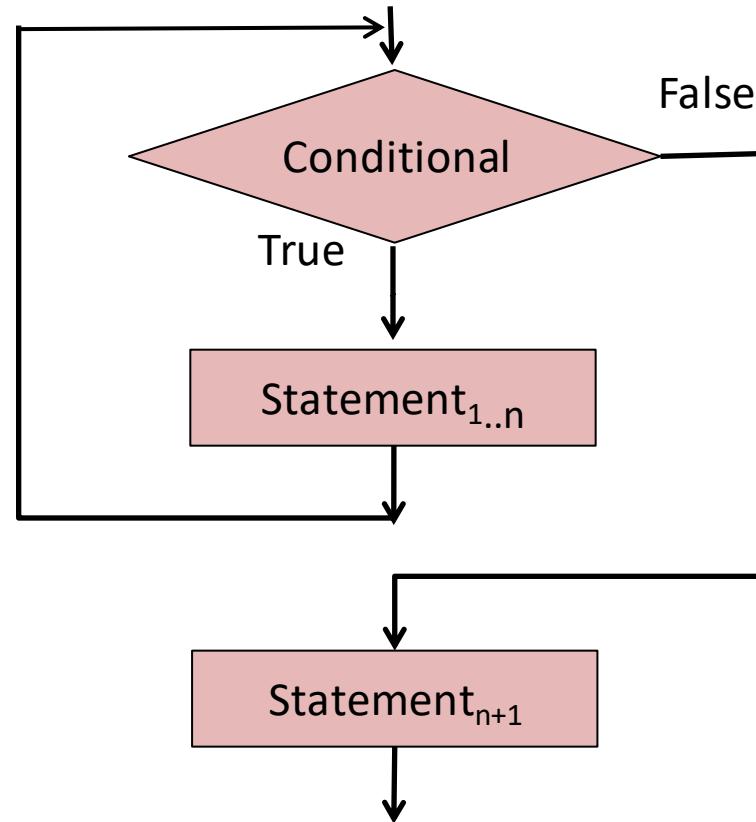
Type hint	Semantics	Examples
Any	<i>Value of any type</i>	v1: Any = 3 v2: Any = "hello world"
Optional[type]	<i>Value of type or None</i>	Optional[int]
List[type]	List of values of type	I1: List[int] = [1, 2, 3] I2: List[str] = ['hello', 'world'] I3: List[List[int]] = [[1, 2], [3, 4, 5]] I4: List[Any] = [1, "two", 3.14]
Sequence[type]	Arbitrary sequence of type	"Hello world" [1, 2, 3] {'one': 1, 'two': 2, }

# While loops

`while` loops offer another way to loop

`while` conditional:

```
statement1..n
statementn+1
```





# Write a function...

... to calculate the sum of a range of integers

```
def sum_range_while(start: int, end: int) -> int:
    """ calculate the sum of the integers starting at
        start and up to, but not including end
    """
    total: int = 0
    while start < end:
        total += start
        start += 1
    return total

assert sum_range_while(1,1) == 0
assert sum_range_while(1,5) == 10
assert sum_range_while(1,2) == 1
```

What if start  $\geq$  end?



# Beware of infinite loops

Programs do exactly what they are told, not necessarily what we want

We may go into an infinite loop if we're not careful with while loops

What happens if I do:

```
i = 0
while i < 5:
    print(i)
```

The while loop prints 0 until you interrupt the program. Why?

Always verify that the conditional becomes False so the loop terminates



# while loops

while loops are more flexible than for loops

But while loops may require more code

```
i = 0
while i < 3:
    print(i)
    i += 1
```

0  
1  
2

```
for i in range(3):
    print(i)
```

0  
1  
2



# When to use `for` and `while`

If you are looping over a sequence, e.g. a range, a list, a string, ...  
then a `for` loop is *probably* best

If you know how many iterations in advance, then a `for` loop is  
*probably* best

If you must loop while some condition is True/False, then a `while`  
loop is *probably* best

If you need a loop with a test in the middle or end, then a `while`  
loop is *probably* best, but a `for` loop may work as well

There's no right or wrong answer. It's just a matter of style.  
Choose the solution that makes your code easier to read,  
understand, and maintain.



# Break out

Sometimes you want to break out of the middle, rather than the top, of a loop

`while conditional:`

`statement1..k`

`if condition:`

`break`

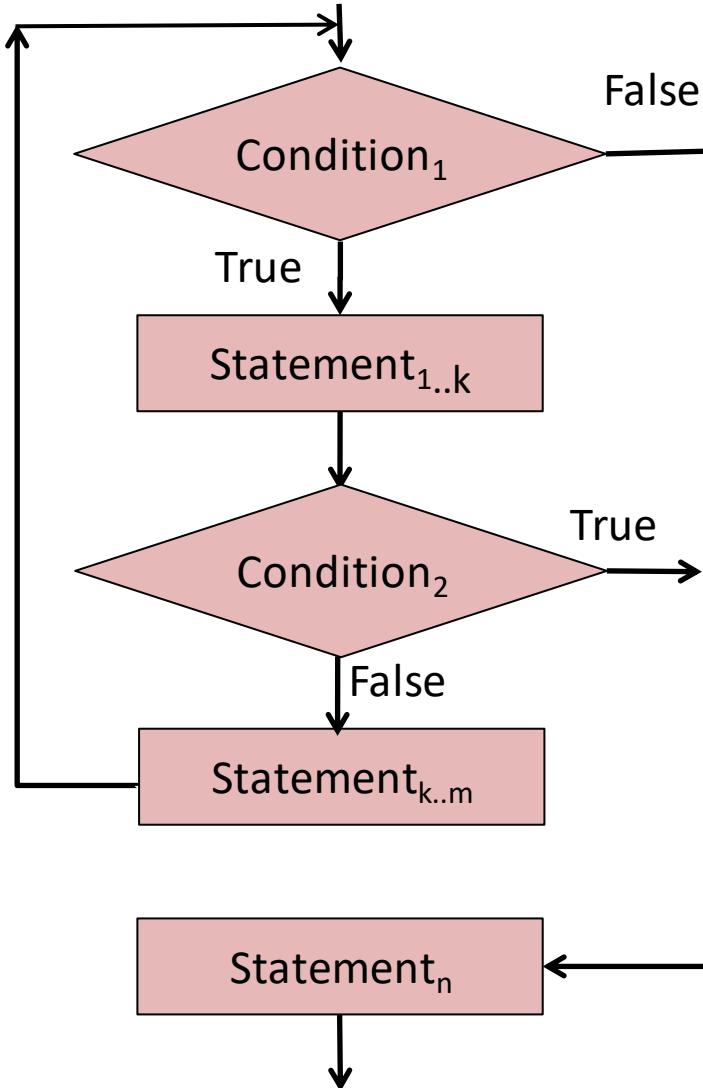
`statementk..m`

`statementn`

`break` leaves the loop and execution moves to the first line following the while block

# break out

```
while condition1:  
    statement1..k  
    if condition2:  
        break  
    statementk..m  
statementn
```





# Controlling the **while** loop

Flags for conditional expressions are one solution but alternatives may be more concise

```
# loop until the user quits  
flag = True  
while flag:  
    if input('Quit?: ') == 'y':  
        flag = False
```

```
# loop until the user quits  
while True:  
    if input('Quit?: ') == 'y':  
        break
```

Choose the alternative works best for the situation

# Where to test?

## Test at the top

*while condition:  
statement<sub>i..j</sub>*

Execute statement<sub>i..j</sub> 0 or  
more times

## Test in the middle

*while True:  
statement<sub>i..j</sub>  
if condition:  
break  
statement<sub>k..l</sub>*

Execute statement<sub>i..j</sub> at  
least once: statement<sub>k..l</sub>  
0 or more times

## Test at the bottom

*while True:  
statement<sub>i..j</sub>  
if condition:  
break*

Execute statement<sub>i..j</sub>  
at least once



# How many digits in a number?

Write a function to calculate the number of digits in a number

e.g. 100 has 3; 2354 has 4; etc.

Use `//` operator – floor division, e.g. `9 // 2 == 4`

Start with the number and continue doing floor division by 10 until we run out of digits

`for` loop or `while` loop?

Don't know the number of iterations in advance so a `while` loop is probably better

# How many digits?

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    total: int = 0  
    while n > 0:  
        total += 1  
        n //= 10  
    return total  
  
assert digits(100) == 3  
assert digits(42) == 2  
assert digits(123456789) == 9  
assert digits(0) == 1
```

---

```
AssertionError                                                 Traceback (most recent call last)  
<ipython-input-28-62dbefc7a27b> in <module>  
      10 assert digits(42) == 2  
      11 assert digits(123456789) == 9  
----> 12 assert digits(0) == 1
```

Test boundary conditions, e.g. 0

```
AssertionError:
```



# Debugging digits()

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    total: int = 0  
    while n > 0:  
        print(f"n={n}")  
        total += 1  
        n //= 10  
    return total  
  
digits(123)
```

The print statement helps with debugging

n=123  
n=12  
n=1

3 digits(123) returns 3



# Testing digits()

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    total: int = 0  
    while n > 0:  
        print(f"n={n}")  
        total += 1  
        n //= 10  
    return total
```

```
digits(0)
```

0

What's wrong? `digits(0) == 0` but should be 1

Wait! Why didn't the print statement print anything?



# Testing digits()

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    total: int = 0  
    while n > 0:  
        print(f"n={n}")  
        total += 1  
        n //= 10  
    return total
```

**digits(0)**

Not entering the loop if  $n == 0$ .  
Just, change ' $>$ ' to ' $\geq$ '  
That will fix it!

Not so fast!!!  
Walk through the logic by hand.  
Will the condition ever be False?

No,  $0 // 10 == 0$   
We're stuck in an infinite loop!  
Time for ^C

0

What's wrong?  $\text{digits}(0) == 0$  but should be 1

Wait! Why didn't the print statement print anything?



# Testing digits()

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    total: int = 0  
    while n > 0:  
        print(f"n={n}")  
        total += 1  
        n //= 10  
    return total
```

How can we fix the problem with 0?

**Wait!** What about negative numbers?  
Work through the logic by hand...  
 $-1 // 10 == -1$  (infinite loop)

Solution is valid only for positive integers!

digits(0)

Add missing test cases for  $<= 0$

0

What's wrong?  $\text{digits}(0) == 0$  but should be 1

Wait! Why didn't the print statement print anything?



# Debugging digits(0)

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    n = abs(n) # absolute value  
    total: int = 1  
    while True:  
        print(f"n={n}")  
        n //= 10  
        if n <= 0:  
            break  
        else:  
            total += 1  
    return total  
  
digits(0)
```

Sometimes we just need to start over  
and try a different approach

Exit from the middle of the loop  
rather than the top

This solution passes all of the  
previous tests...

n=0

1



# How many digits? Pythonic solution

```
def digits(n: int) -> int:  
    """ return the number of digits in n """  
    return len(str(abs(n)))  
  
assert digits(100) == 3  
assert digits(42) == 2  
assert digits(0) == 1  
assert digits(-32) == 2
```

Consider alternative solutions then choose the “best”

Readability/maintainability

Consistency

Being “too clever” may cause problems



# break out of for loop

```
# print the even numbers
for num in [2, 8, 7, 4, 9]:
    if num % 2 == 1: # an odd number
        break      # stop at the first odd
    print(num)

print('Done')
```

2  
8  
Done

Breaking out of a **for** loop is fairly unusual

Think about a different solution before committing  
to breaking out of a **for** loop



# while/else, for/else

Python offers `while/else` and `for/else` statements

The `else` clause is executed if the loop terminates naturally,  
**without** `break`

```
def char_in_string(target: str, string: str) -> None:
    for ch in string:
        if ch == target:
            print("Breaking out")
            break
    else: # execute this block if did not break
        print("Not found")
    print("Next statement")

char_in_string('y', 'Python')
```

Breaking out

Next statement



# Using for/else

The `else` clause is executed if the loop terminates naturally, **without** `break`

```
for i in range(2, 20):
    for n in range(2, i):
        if i % n == 0:
            break
    else:
        print(f"{i} is prime") # if here, then i is prime
```

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```



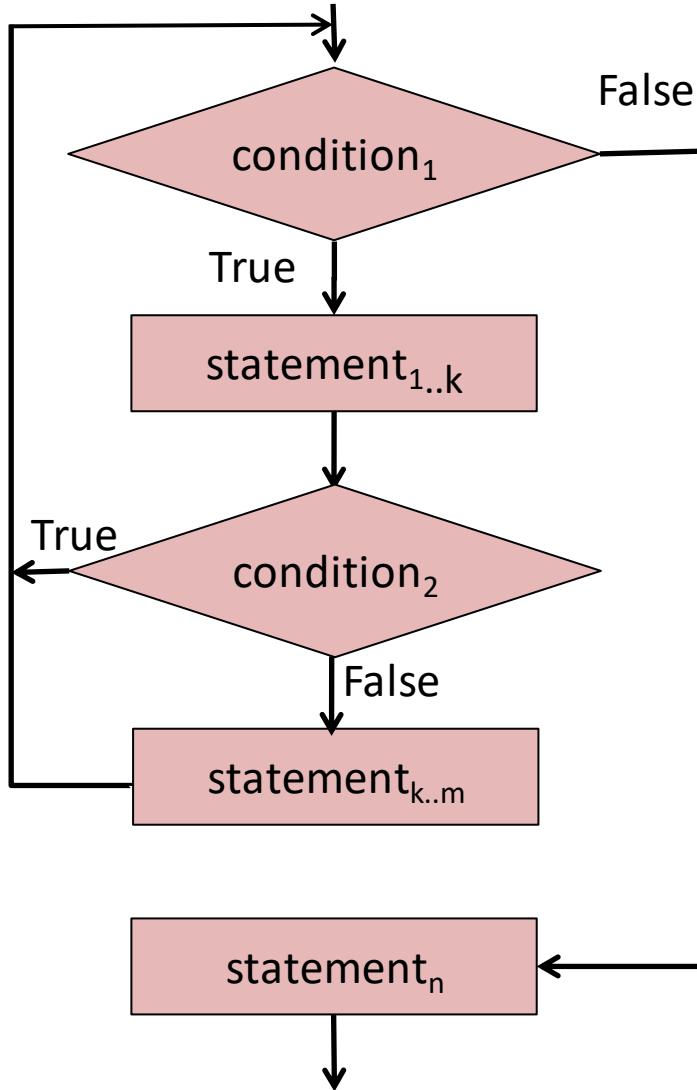
# continue

`break` exits the loop

`continue` skips the rest the loop and continues at the next iteration

# continue

```
while condition1:  
    statement1..k  
    if condition2:  
        continue  
    statementk..m  
statementn
```





# Continue to next item in `for` loop

```
# print the even numbers, skip the odd numbers
for num in [2, 8, 7, 4, 9]:
    if num % 2 == 1: # an odd number
        continue # skip rest of the block
                    # process next item in the loop
    print(num)

print('Done')
```

```
2
8
4
Done
```

Think about a different solution before committing to `break` or `continue`, but sometimes `break/continue` is best



# Nested loops

Nested loops are very common

```
for row in range(3):
    for col in range(3):
        print(f"{row} * {col} = {row * col}")
```

```
0 * 0 = 0
0 * 1 = 0
0 * 2 = 0
1 * 0 = 0
1 * 1 = 1
1 * 2 = 2
2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
```





# Generators

Generators create custom sequences with ***lazy evaluation***

The generator creates the next item in the sequence ***on demand***  
Don't create the next item until requested

Consider a system that collects data from an internet feed,  
e.g. the Twitter feed for Python tweets

```
for tweet in twitter_feed():
    tweet.analyze()
```

We could read N tweets in advance and store them in a list or we  
can use a generator to get the next tweet

Generators are great for co-routines

```
cat file | filter1 | filter2 | filter3
```



# Writing generator functions

A generator looks just like any other function, except...

Generators include one or more `yield` expressions

```
from typing import Iterator
```

Yield a sequence of int values

```
def simple_generator() -> Iterator[int]:
```

```
    print("before yield 7")  
    yield 7
```

Yield 7 and wait for the next call

```
    print("before yield 4")  
    yield 4
```

Yield 4 and wait for the next call

```
    print("before yield 3")  
    yield 3
```

Yield 3 and wait for the next call

```
for i in simple_generator():  
    print(f"i = {i}")
```

The generator ends on implicit or  
explicit return

```
before yield 7  
i = 7  
before yield 4  
i = 4  
before yield 3  
i = 3
```



# Writing generator functions

```
def fib(n: int) -> Iterator[int]:  
    """ generate the first n Fibonacci numbers """  
    prev2: int = 0  
    prev1: int = 1  
    for i in range(n):  
        yield prev2  
        prev2, prev1 = prev1, prev2 + prev1
```

Yield "returns" the value, but execution continues here on next call

```
for i in fib(6):  
    print(i)
```

The generator exits on implicit or explicit return

0  
1  
1  
2  
3  
5

Parameters and local variables are maintained across subsequent yields until the function returns



# Running generator functions

```
def fib(n: int) -> Iterator[int]:
    """ generate the first n Fibonacci numbers """
    prev2: int = 0
    prev1: int = 1
    for i in range(n):
        print(f"fib(): about to yield {prev2}")
        yield prev2
        print(f"fib(): following yield {prev2}")
        prev2, prev1 = prev1, prev2 + prev1

print("main: about to call fib()")
for i in fib(6):
    print(f"main: consuming {i} from fib()")
print("all done")
```

```
main: about to call fib()
fib(): about to yield 0
main: consuming 0 from fib()
fib(): following yield 0
fib(): about to yield 1
main: consuming 1 from fib()
fib(): following yield 1
fib(): about to yield 1
main: consuming 1 from fib()
fib(): following yield 1
fib(): about to yield 2
main: consuming 2 from fib()
fib(): following yield 2
fib(): about to yield 3
main: consuming 3 from fib()
fib(): following yield 3
fib(): about to yield 5
main: consuming 5 from fib()
fib(): following yield 5
all done
```



# Yield from any sequence

```
from typing import Sequence, Any, Iterator

def seq_gen(s: Sequence) -> Iterator[Any]:
    yield from s

for item in seq_gen('Hi'):
    print(item)
```

H  
i

```
for item in seq_gen([1, 2, 3]):
    print(item)
```

1  
2  
3

```
for item in seq_gen(range(5, 7)):
    print(item)
```

5  
6

This works for  
sequences of any type,  
e.g. strings, lists, tuples,  
...



# More type hints

```
from typing import Sequence, Iterator, Any
```

Type hint	Semantics	Examples
Sequence[ <i>type</i> ]	Arbitrary sequence of type	“Hello world” [1, 2, 3] {‘one’: 1, ‘two’: 2, }
Iterator[ <i>type</i> ]	A generator yielding a sequence of <i>type</i>	def integers() -> Iterator[int]: # generate a sequence of int values

Example:

```
def my_enumerate(seq: Sequence[Any]) -> Iterator[Any]:
```

Given a sequence of Any type, return a generator that yields value of Any type



# Write a generator...

... that counts from 0 up to a limit and then back to 0

e.g. 0, 1, 2, 3, 2, 1, 0

How to start?

How to count up to limit?

`range(0, limit + 1)`

Consume all values one by one  
then move to next statement

How to count down limit to 0?

`range(limit - 1, -1, -1)`

Wait... we can simplify that a little...

`range(limit) # 0 to n - 1`

`range(limit, -1, -1) # n down to 0`



# Write a generator...

```
def updown(n: int) -> Iterator[int]:  
    """ count up to n and back down to 0 """  
    yield from range(n) # 0 to n-1  
    yield from range(n, -1, -1) # n to 0
```

```
for i in updown(2):  
    print(i)
```

Consume all values one by one  
then move to next statement

0  
1  
2  
1  
0

# Unbounded generators

```
def integers() -> Iterator[int]:  
    """ An infinite sequence of integers """  
    i = 0  
    while True: # we can't use a for loop  
        yield i  
        i += 1  
  
# using an unbounded generator  
limit: int = 3  
for i in integers():  
    print(i)  
    limit -= 1  
    if limit <= 0:  
        break
```

Yield the next integer for as long as anyone calls `__next__()`

Need to limit the number of calls to avoid infinite loop

0  
1  
2



# Generator next()

```
def integers() -> Iterator[int]:  
    """ An infinite sequence of integers  
    i = 0  
    while True: # we can't use a for loop  
        yield i  
        i += 1  
  
# using an unbounded generator  
gen: Iterator[int] = integers()  
for i in range(3):  
    print(next(gen)) # next(gen) gets the next value
```

Wait!!! while True with no break? That's an infinite loop!?!?

next() is called only when next value is needed, so yes, it's an infinite loop, but not a problem

0  
1  
2

next(generator) gets the next value from a generator

# When to use generators?

Any time you need a sequence

Generators are more efficient than prepopulating a list

While waiting for asynchronous events

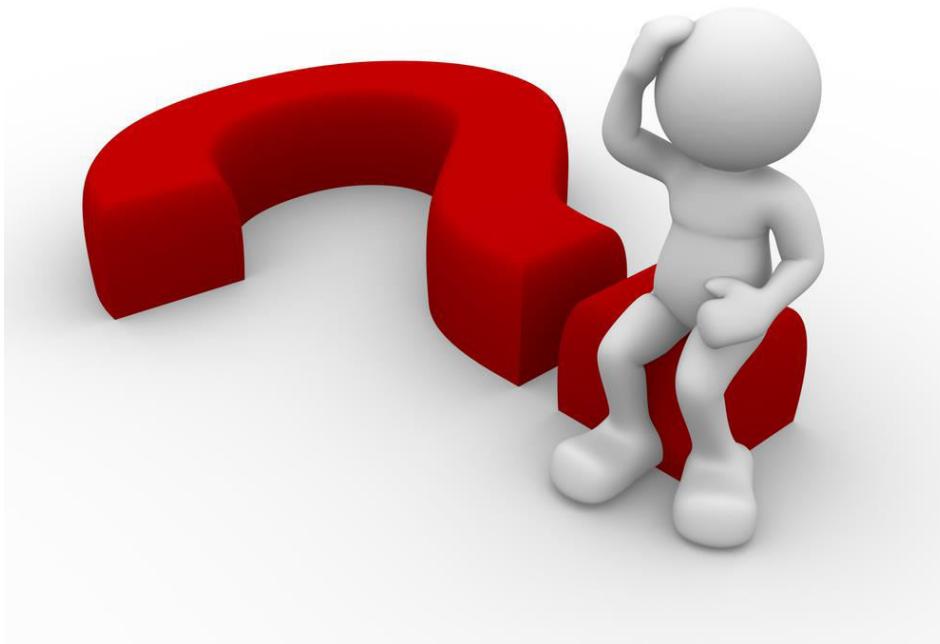
Any time lazy evaluation makes sense

e.g. very large sequences

Any time you may need to read multiple lines from a file  
to retrieve a single entry



# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Strings, Slices, Files, and Coding Style Guidelines*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from:

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

<https://docs.python.org/3/library/stdtypes.html>

<https://mkaz.tech/python-string-format.html>

# Today's topics

Strings

Slices

Working with files

Working with JSON

Reading from the Web

Coding style guidelines



# Text processing with Python

Python is exceptionally good at data processing

Applications include:

Data Science/Big Data

Data Engineering

Natural Language Processing

Machine Learning



Very powerful string manipulation features



# Strings

Strings are compound data types

A **string** data type is made up of **character** data types

Use the string as an object or use its parts (the characters)

Strings have many useful operations

```
s:str = "Hello world!"  
s.upper() # apply the upper() method to s
```

```
'HELLO WORLD! '
```

```
s.split() # split the string into tokens
```

```
['Hello', 'world!']
```

```
s[0:5]
```

```
'Hello'
```



# Comparing strings

Comparison operators work as expected

```
'banana' < 'bat'
```

True

```
'banana' == 'banana'
```

True

```
'BaNaNa'.lower() == 'bAnAnA'.lower()
```

True

The string is not changed:  
lower() returns a new string

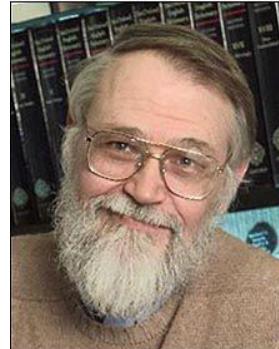
You may want to convert strings to lower case when comparing and/or counting word occurrences so 'YES', 'Yes', and 'yes' are the same word

# Python string manipulation

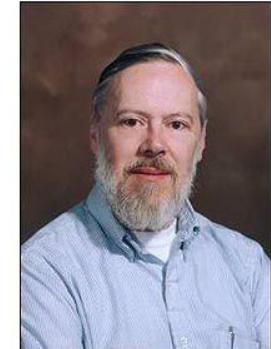
## Concatenation

```
'Hello ' + 'world'
```

```
'Hello world'
```



Brian Kernighan



Dennis Ritchie

## Replication

```
'Ho' * 3
```

```
'HoHoHo'
```



VectorStock®

VectorStock.com/12028813



# Strings and iterators

Recall you can use strings as sequences that can be used with iterators

```
for c in "Python":  
    print(c)
```

P  
y  
t  
h  
o  
n



# Subsets of strings with slices

## Choose a subset of a string with slices

```
s:str = 'Hello world'  
s[1] # start counting at 0
```

```
'e'
```

```
s[0:5] # chars 0-4
```

```
'Hello'
```

```
s[6:12] # chars 6-11
```

```
'world'
```

```
s[6:] # chars 6 through the end
```

```
'world'
```

```
s[:5] # chars 0-4
```

```
'Hello'
```

Slices work the same way  
with other sequences,  
e.g. lists, tuples



# Subsets of strings with slices

The slice index may also be negative (from the end)

Think of a negative index  $i$  as  $\text{len}(s) - i$

```
s:str = 'Hello world'  
s[-1] # s[len(s)-1]
```

'd'

```
s[-5:] # s[len(s)-5:] -> the last 5 chars
```

'world'

```
s[:-6] # all but the last 6 characters
```

'Hello'

```
s[-5:-3] # s[len(s)-5:len(s)-3] == s[6,8]
```

'wo'



# Slices of sequences

[ *start* : *end* : *step* ]

***start*** - the offset from the beginning (offset from 0)

***end*** – up to, but not including end, i.e. end – 1

***step*** – skip every ***step*** items

***start*, *end*, and *step*** arguments are optional



# Reverse a string with slices

Problem: How to reverse a string using only slices

Recall `slice[start : end : step]`

We want the entire string, but we want to step from the end back to the beginning

```
s:str = 'Hello world'  
s[::-1] # gets the entire string
```

```
'Hello world'
```

```
s[::-1] # reverse the string
```

```
'dlrow olleH'
```

Be careful about trading cleverness for readability!



# Strings are immutable

```
s:str = 'Hello world'  
s[0]
```

```
'H'
```

```
s[0] = 'J' # try to change "Hello world" to "Jello world"
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-3-c5726cf43a96> in <module>  
----> 1 s[0] = 'J' # try to change "Hello world" to "Jello world"
```

```
TypeError: 'str' object does not support item assignment
```



# Are you in there?

Check for substrings with ***in*** and ***not in***

```
s:str = 'Hello world'  
'ello' in s
```

True

```
'o' in s
```

True

```
'L' in s
```

False

```
'L' not in s
```

True



# Count characters in a string

```
def count(ch:str, string:str) -> int:  
    """ Return the number of occurrences of ch in str """  
    cnt: int = 0  
    for c in string:  
        if c == ch:  
            cnt += 1  
    return cnt  
  
s:str = "Hello world!"  
assert count('H', s) == 1  
assert count('l', s) == 3  
assert count('x', s) == 0  
assert count('x', '') == 0
```

We can also use Python's  
string method  
s.count('H')

```
s.count('l')
```

# No more vowels

```
def remove_vowels(s:str) -> str:  
    vowels: str = 'aeiou'  
    new_string:str = ''  
    for ch in s:  
        if ch.lower() not in vowels:  
            new_string += ch  
    return new_string
```

```
s:str = "Hello world!"  
remove_vowels(s)
```

```
'Hll wrld!'
```

Use `ch.lower()`, not `s.lower()`, so the returned string matches the upper/lower case characters in `s`

Can use same approach to remove punctuation!  
`string.punctuation` is string with all punctuation

```
s
```

```
'Hello world!'
```

Returned a copy of the string – didn't change the string `s`



# Find a character in a string: slices

```
def find_char(ch:str, s:str) -> int:  
    """ Find the first occurrence of ch in str  
        Return -1 if not found  
    """  
  
    for i in range(len(s)):  
        if s[i] == ch:  
            return i  
    return -1  
  
s:str = "Hello world!"  
assert find_char('H', s) == 0  
assert find_char('!', s) == 11  
assert find_char('x', s) == -1
```



# Find a character in a string: enumerate

```
def find_char(ch:str, s:str) -> int:  
    """ Find the first occurrence of ch in s  
        Return -1 if not found  
    """  
  
    for i, c in enumerate(s):  
        if c == ch:  
            return i  
    return -1
```

Consider both slices and  
enumerate() alternatives

```
s:str = "Hello world!"  
assert find_char('H', s) == 0  
assert find_char('!', s) == 11  
assert find_char('x', s) == -1
```



# String.find(target)

Search for a target string within a string

Return the offset where the target string starts or -1

```
'hello'.find('he') # find substrings
```

0

```
'hello'.find('o')
```

4

```
'hello'.find('not there')
```

-1

Recall that we're calling the 'find' method on string 'hello' with the argument('he')



# Splitting strings into tokens

Read a line and process each token in the line

```
groceries:str = "eggs milk bread"  
for item in groceries.split():  
    print(item)
```

eggs  
milk  
bread

string.split() with no argument defaults the separator to whitespace

```
from typing import List  
  
header:str = "firstName|lastName|age"  
fields>List[str] = header.split('|')  
fields  
  
['firstName', 'lastName', 'age']
```

string.split(arg) splits on the specified argument

# Splitting strings into tokens: maxsplit

Split a string into  $n$  components

```
name:str = "Thorton Billy Bob"  
name.split()
```

```
[ 'Thorton', 'Billy', 'Bob' ]
```

```
name.split(maxsplit=2)
```

```
[ 'Thorton', 'Billy', 'Bob' ]
```

```
name.split(maxsplit=1)
```

```
[ 'Thorton', 'Billy Bob' ]
```





# String length

```
len('Hello')
```

5

```
'Hello'.len()
```

```
----> 1 'Hello'.len()
```

```
AttributeError: 'str' object has no attribute 'len'
```

Python provides built in `len()` function for many types  
which is more efficient in time and space

(But it's an exception that you must remember)



# string.join(seq)

string.split(delim) splits a string into a list of tokens

string.join(seq) creates a string from a list of tokens separated by string

```
from typing import List  
  
groceries:List[str] = ["eggs", "milk", "bread"]  
" | ".join(groceries)  
  
'eggs|milk|bread'
```

```
", ".join(groceries)
```

```
'eggs, milk, bread'
```

These string methods are used frequently!



# string.join(seq)

Given a list of characters, how can we create a string?

```
from typing import List  
  
chars:List[str] = ['P', 'y', 't', 'h', 'o', 'n']  
"".join(chars)  
  
'Python'
```

Note: seq must be a sequence of str



# String Methods

String class supports a number of useful methods

String Method	Semantics
<code>string.startswith(s)</code>	returns True if string begins with s
<code>string.endswith(s)</code>	returns True if string ends with s
<code>string.split([sep=white space], [maxsplit=-1])</code>	Split a string into tokens
<code>string.strip()</code>	strips leading and trailing whitespace
<code>string.lstrip()</code>	strips leading whitespace
<code>string.rstrip()</code>	strips trailing whitespace

See <https://docs.python.org/3/library/stdtypes.html>



# string.format()

string.format() provides a very powerful tool to create and format strings

Frequently used in print() statements, but many uses

Placeholders insert values into the string

```
"{0} + {1} == {2}" .format(3, 4, 7)
```

'3 + 4 == 7'

If you don't specify an argument offset within the placeholder, then the arguments are used sequentially

```
"{} + {} == {}".format(3, 4, 7)
```

'3 + 4 == 7'



# f-strings simplify string.format()

f-strings are a syntactic shortcut for `str.format()`

f-strings were first offered in Python 3.6

Expressions within `{expr}` are replaced with the `expr`

```
num = 3  
denom = 4  
f'{num}/{denom}'
```

Don't forget the preceding 'f'!

'3/4'  
`expr` can be any valid Python expression

f-strings make your code much easier to write and read



# string.format()/f-string examples

```
"{0} {1} {1} and a bottle of rum".format('Yo', 'ho')
```

```
'Yo ho ho and a bottle of rum'
```

```
f"pi={3.1415926:.2f}"
```

```
'pi=3.14'
```

```
f"Numbers with commas: {1234567:,}"
```

```
'Numbers with commas: 1,234,567'
```

```
f"Left aligned field with width 5 | {2:<5d} | "
```

```
'Left aligned field with width 5 | 2 | '
```

<https://mkaz.tech/python-string-format.html> is a great site to see the many options



# Chaining methods in a single call

We've seen that we can invoke class methods with  
`obj.method()`, e.g. `'hello world'.split()`

We can chain several calls together in a single statement

```
"Hello world".strip(' ').lower().split()  
['hello', 'world']
```

Is equivalent to

```
s = "Hello world"  
s = s.lower()  
s.split()  
  
['hello', 'world']
```

When chaining methods, the methods are applied left to right



# Detecting palindromes

A string is a palindrome if the string is the same forward and backward

$\xrightarrow{\hspace{1cm}}$   
AbCdCbA  
 $\xleftarrow{\hspace{1cm}}$

Two different approaches

1. Compare the string the reverse of the string: if string == reversed string then the string is a palindrome
2. Start at beginning and move to the right and at the end of string and move left, comparing subsequent characters until either the characters don't match or you meet in the middle



# Palindrome detection: reverse

AbCdCbA

```
def is_palindrome_reverse(s:str) -> bool:  
    return s == reverse(s)  
  
assert is_palindrome_reverse('abcdcba') is True  
assert is_palindrome_reverse('abcdcbae') is False
```

Note: reverse(string) is a homework assignment

This solution is very simple to implement and understand

It's a great solution for short strings but...

Not very efficient



# Palindrome detection: compare

AbCdCbA  
↑           ↑  
left → ← right

```
def is_palindrome_fast(s:str) -> bool:  
    left:int = 0                                   # offset of first char  
    right:int = len(s) - 1                        # offset of last char  
    while left < right:  
        if s[left] != s[right]:  
            return False    # found mismatch  
        else:  
            left += 1  
            right -= 1  
    return True    # left passed right so palindrome  
  
assert is_palindrome_fast('abcdcba') is True  
assert is_palindrome_fast('abcdcbae') is False
```

# Compare efficiencies

Considerations:

How efficient is each algorithm?



How quickly can we determine if a string is a palindrome?

What if the string is very long (a book)?

How frequently do we need to use this feature?

How hard to write and maintain both versions?

No “correct” answer for all situations



# Summary

Strings are a fundamental data type in Python

There are many string functions to choose from

The most important include:

in, not in

Slices

find()

format()

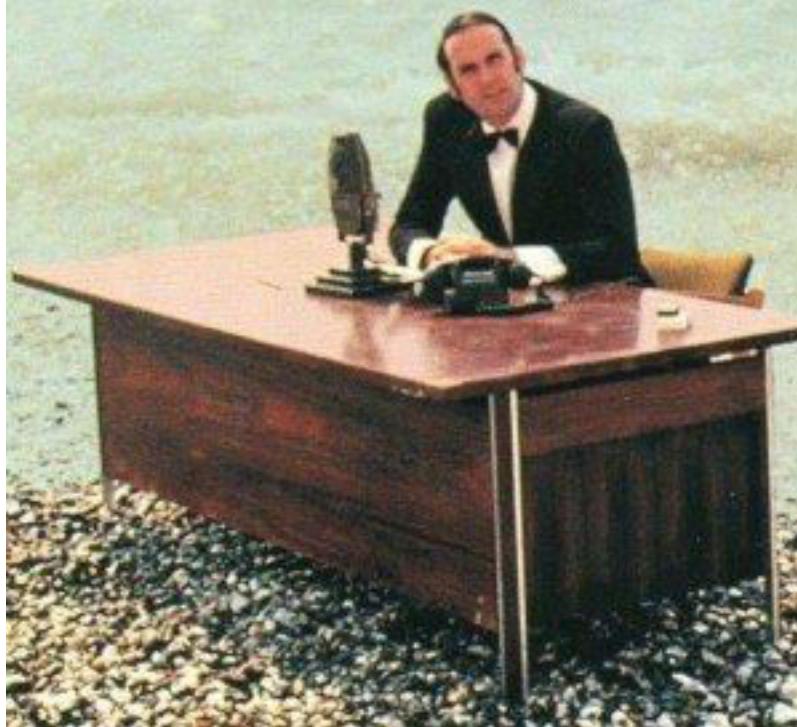
f-strings

split()

strip()

join()

And now  
for something  
completely different...





# Working with files

Steps for reading from a file:

1. Get file name
2. Open the file
3. Read the file
  - a. Line by line
  - b. All at once into a list of lines
  - c. All at once into a buffer
4. Close the file



# README file

- 1 Hello world!
- 2 Goodbye world!
- 3 See you later!
- 4 Go Ducks!
- 5

Trailing \n



# Reading a text file line by line

```
from typing import IO

file_name:str = 'README'
try:
    fp:IO = open(file_name, 'r')
except FileNotFoundError:
    print(f"Can't open {file_name}")
else:
    with fp:
        for line in fp: # fp moves line by line
            # process the line
            print(line.strip())
```

Hello world!  
Goodbye world!  
See you later!  
Go Ducks!

Strip the line to remove  
the trailing \n



# Reading a text file into a list of lines

```
from typing import IO, List

file_name:str = 'README'
try:
    fp:IO = open(file_name, 'r')
except FileNotFoundError:
    print(f"Can't open {file_name}")
else:
    with fp:
        lines:List[str] = fp.readlines() # lines is a list
        print(f"{file_name} has {len(lines)} lines")
        print(f"Line 2: -->{lines[1]}<--")
```

readlines() returns a list of strings, one item for each line in the file

README has 5 lines  
Line 2: -->Goodbye world!  
<--

Each line ends with a \n



# Reading a text file into a buffer

```
from typing import IO, List

file_name:str = 'README'
try:
    fp:IO = open(file_name, 'r')
except FileNotFoundError:
    print(f"Can't open {file_name}")
else:
    with fp:
        string:str = fp.read() # read the entire file
        print(f"{file_name} has {len(string)} characters")
        freq:int = string.count('world')
        print(f"'world' occurs {freq} times")
```

README has 54 characters  
'world' occurs 2 times

Line breaks are stored as '\n'

string

'Hello world!\nGoodbye world!\nSee you later!\nGo Ducks!\n\n'



# Writing to a text file

```
from typing import IO

log_file:str = 'sample_output.txt'
try:
    fp:IO = open(log_file, 'w')
except FileNotFoundError:
    print(f"Can't open {log_file} for writing")
else:
    with fp:
        for i in range(3):
            fp.write(f"line {i}\n")
```

'w' creates a new file if the file doesn't exist or overwrites the file if it does exist

'a' creates a new file if the file doesn't exist or appends to the file if it does exist

```
jrr ~ $ cat sampleOutput.txt
line 0
line 1
line 2
```

Include '\n' unless you want everything on one line



# Writing JSON files

## JSON is commonly used by web applications

```
from typing import IO, Dict, Any
import json

JSON_file:str = 'sampleJSON.txt'
data:Dict[str, Any] = {'name' : 'Nanda', 'Age' : 22, 'GPA' : 3.97}
try:
    fp:IO = open(JSON_file, 'w')
except FileNotFoundError:
    print(f"Can't open '{JSON_file}' for writing")
else:
    with fp:
        print(f"Data: {data}")
        print(f"Data as JSON: {json.dumps(data)}")
        json.dump(data, fp) # write data to fp as JSON
```

Data: {'name': 'Nanda', 'Age': 22, 'GPA': 3.97}

Data as JSON: {"name": "Nanda", "Age": 22, "GPA": 3.97}



# Reading JSON files

```
jrr ~ $ cat sampleJSON.txt  
{"Age": 22, "GPA": 3.97, "name": "Nanda"}
```

```
from typing import IO, Dict, Any  
import json  
  
JSON_file:str = 'sampleJSON.txt'  
try:  
    fp:IO = open(JSON_file, 'r')  
except FileNotFoundError:  
    print(f"Can't open '{JSON_file}' for reading")  
else:  
    with fp:  
        data:Dict[str, Any] = json.load(fp)  
        print(f"Read from JSON file: {data}")
```

```
Read from JSON file: {'name': 'Nanda', 'Age': 22, 'GPA': 3.97}
```



# Reading from the Web

Python supports reading arbitrary URLs from the Web

```
import urllib.request
from typing import IO, Dict, Any

url:str ="http://www1.ncdc.noaa.gov/pub/data/cdo/samples/PRECIP_HLY_sample_csv.csv"
try:
    fp:IO = urllib.request.urlopen(url)
except ValueError:
    print(f"Can't open {url}")
else:
    with fp:
        for line in fp:
            print(line)

b'STATION,STATION_NAME,ELEVATION,LATITUDE,LONGITUDE,DATE,HPCP,Measurement Flag,Quality Flag
\n'
b'COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 00:00,99999,,\n'
b'COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 01:00,0,g,\n'
b'COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100102 06:00,1, ,\n'
```

# Reading from the Web

Several good solutions for reading from the web

Urllib – open arbitrary resources by URL

Requests – HTTP for Humans

BeautifulSoup4 – parses data from HTML and XML files

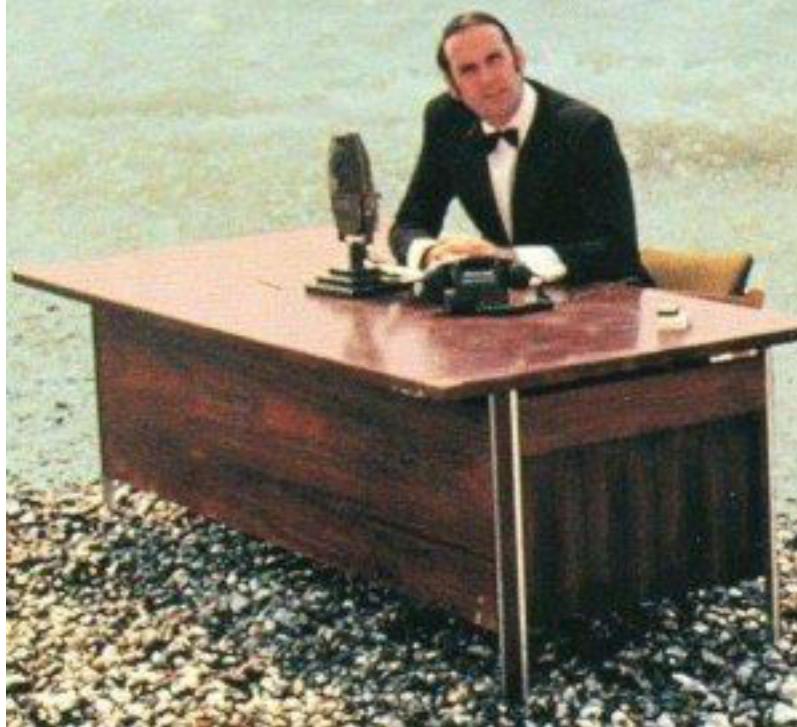
Scrapy – scraping and web crawling

Good solutions for parsing HTML, JSON, XML, etc.

Tweepy - reads from the Twitter feed



And now  
for something  
completely different...





# Coding style guidelines

We spend far more time reading than writing code

Following coding style guidelines/standards make code easier to read and understand

Same Code: Which would you rather read?

```
def fib(n):
    """
    Return the nth element of the
    Fibonacci sequence
    """
    if n == 1 or n == 2:
        return 1
    else:
        prev2, prev1 = 1, 1

        while n > 2:
            prev2, prev1 = prev1, prev1 + prev2
            n -= 1
    return prev1
```

```
def fib(n)      :
    if ((n==1 )
    or (n ==  2)): return 1
    else:
        prev2,prev1=1,1
        while n> 2 :
            prev2,prev1=prev1,prev1+prev2;n-=1
    return prev1
```



# Python Coding Style Guidelines

Coding style guidelines provide consistency

Coding style guidelines make code easier to read and understand:

- Encourage consistency across a project or organization

- Help the reader to recognize relevant chunks of code

  - Consistent use of white space

  - Consistent formatting of program elements

- Avoid developing and using bad habits

If your project has a coding style then follow it

Otherwise adopt an existing coding style

Google: <http://google.github.io/styleguide/pyguide.html>

Python: <https://www.python.org/dev/peps/pep-0008/>



# Python Style: Bad vs Good

```
class MyGiganticUglyClass(object):
    def iUsedToWriteJava(self, x, y = 42): ❶ ❷ ❸
        blnTwoSpacesAreMoreEfficient = 1 ❶ ❸ ❹
        while author.tragicallyConfused(): ❺
            print "Three spaces FTW roflbbq!!!" ❻
        if (new_addition): ❺
            four_spaces_are_best = True ❻
        if (multipleAuthors \ ❺ ❻
            or peopleDisagree): ❺
            print "tabs! spaces are so mainstream"
        ...
        return ((pain) and (suffering)) ❺
```

1. Indentation issues
2. White space issues
3. Inconsistent case
4. Hungarian Notation
5. Extraneous parens
6. Extraneous line continuations

```
class MyMorePythonicClass(object):
    def now_i_write_python(self, x, y=42):
        two_spaces_hamper_readability = True

        while author.tragically_confused(): ❶
            print "Three spaces? What was I thinking?"

        if new_addition:
            four_spaces_are_best = True

        if (multiple_authors or
            people_disagree): ❷
            print "Mixing tabs and spaces is dangerous!"

        ...
        return sunshine and puppies
```

Source: How to Make Mistakes in Python  
by Mike Pirnat  
Copyright © 2015 O'Reilly Media, Inc. All rights reserved.



# Sample Guidelines

Indentation: use 4 spaces per indentation level.

Blank lines:

- Use blank lines sparingly in functions to indicate logical sections

- Use two blank lines between top level function and class definitions

- Use one blank line inside a class to separate method definitions

String quotes: use either single or double quotes consistently

Avoid multiple statements per line

Docstrings should be used for all public modules, functions, classes, and methods

Avoid global variables

- Global constants are okay

Use parens sparingly

Explicitly close every file you open



# Return Statement Guidelines

Be consistent: if any return statement returns an expression, than all return statements should return an expression, including an explicit return None

Yes:

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

No:

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)

def bar(x):
    if x < 0:
        return
    return math.sqrt(x)
```

What if  $x \leq 0$ ?



# Naming Guidelines

Object	Naming Guideline	Example
Packages, Modules, Variables, Parameters, Local vars	lower_case_with_underscore	my_package my_module running_total
Classes, Exceptions	CapitalWords	ExceptionHandler Mammal
Functions, Method names	lower_case_with_underscore()	foo_bar()
Constants	CAPS_WITH_UNDERSCORE	MAX_INT

# Coding guidelines summary

Coding guidelines improve code readability and maintainability

Don't require a lot of effort

Coding guidelines are all about being consistent

Within your code

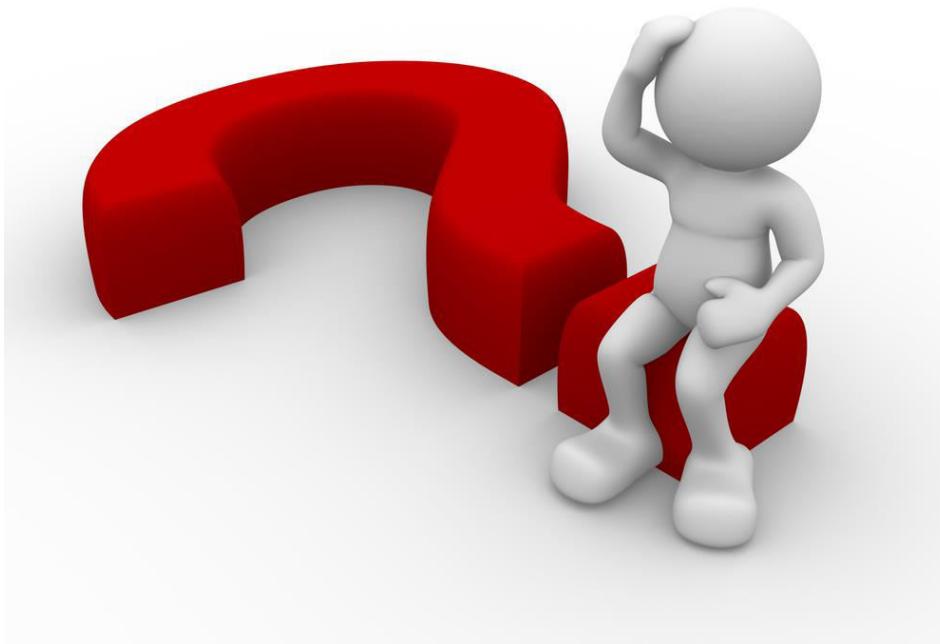
Within your organization

May seem to be much ado about nothing

Low cost, high reward



# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Container Classes: Lists*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

And

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

<https://docs.python.org/3/library/stdtypes.html>

# Today's topics

## Container Classes

Lists, tuples, dictionaries, sets

### Lists

Creating lists

List methods

When things go wrong

List comprehensions

### Improving algorithm performance

Speed/Space tradeoff





# Containers

Python provides several different types of containers

**Lists** - arbitrary values and mutable - [1, 2, 3]

**Tuples** - like a list, but immutable - (1, 2, 3)

**Dictionaries** – Key/Value pairs – {'one':1,'two':2,'three':3}

**Sets** – {1, 2 3}

- Like lists, but with only unique values
- Like lists, but unordered
- Like dictionaries with keys but no values

These containers are fundamental in Python and are used in most Python programs



# Lists

Lists are an **ordered** collection of values of potentially different types, including numbers, strings, lists, tuples, sets, ...

```
from typing import List, Any
```

Any matches any type

```
lst:List[Any] = [1, 'two', 3.14, ['sublist', 4]]
```

Lists are surrounded by **square brackets** [ ]

The values can be any Python object, e.g. int, float, str, list, dict, instance of class, ...



# Create a list

```
lst1: List[Any] = list()      # create an empty list
lst2: List[Any] = []          # create another empty list
lst3: List[str] = ['uno', 'dos', 'tres'] # initialize a list
lst4: List[Any] = [1, 'two', 3.14, ['sublist', 4]]
```

## Some functions return lists

```
'Hello world!'.split()
```

```
['Hello', 'world!']
```

```
import os
os.listdir('/Users/jrr/Downloads/courses')
```

```
['567', '810', '555', '540']
```



# Creating a list from a sequence

`list()` takes an optional argument which must be a sequence, e.g. string, range, list, dict, or set.

The new list contains the elements in the sequence, **not** the sequence itself

```
list('hello')
```

Careful! Not ['hello']

```
['h', 'e', 'l', 'l', 'o']
```

```
lst3:List[int] = [1, 2, 3]  
list(lst3)
```

Create a list with the 3 elements in lst3

```
[1, 2, 3]
```

```
[lst3]
```

Create a list with 1 element, lst3

```
[[1, 2, 3]]
```



# Accessing list elements with index and slices

Index and slices work the same with lists as with strings

```
lst>List[Any] = [1, 'two', 3.14, ['sublist', 4]]  
lst[1] # recall slice offsets start with 0
```

```
'two'
```

```
lst[0:3]
```

```
[1, 'two', 3.14]
```

```
lst[3][1]
```

```
4
```



# Lists of Lists

How can we use lists to store multidimensional arrays?

How can I store the following matrix in Python?

0	1	2
3	4	5
6	7	8

```
matrix:List[List[int]] = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

```
matrix[1][1] = 9
```

```
matrix[1][1] # retrieve the value
```



# Modify lists with an index

Lists are ***mutable***, i.e. they can be changed

```
lst:List[Any] = [1, 2, 3]
lst[1]
```

2

```
lst[1] = 'two'
lst[1]
```

'two'

```
lst
```

```
[1, 'two', 3]
```



# Modify list slice with an iterable

A list slice may be modified with an iterable

```
lst:List[Any] = [1, 2, 3]  
lst[0:2]
```

```
[1, 2]
```

```
lst[0:2] = 4
```

**TypeError:** can only assign an iterable

```
lst[0:2] = [4, 5, 6, 7]
```

```
lst
```

The replacement ***may*** be a different size, but it ***must*** be an iterable

# Lists are mutable: add items

list.append(), list.extend(), +=

```
lst:List[Any] = list() # an empty list
lst.append(1) # append an element to the end
lst.append('two')
lst.append([3, 4])
lst
```

```
[1, 'two', [3, 4]]
```

```
lst.extend([5, 6]) # concatenate the two lists
lst
```

```
[1, 'two', [3, 4], 5, 6]
```

```
lst += [7, 8, 9]
lst
```

```
[1, 'two', [3, 4], 5, 6, 7, 8, 9]
```





# Careful with appending strings to lists

list.extend() and += may surprise you

Recall that a string is a sequence with an iterator

```
lst:List[Any] = [1]
lst.append('two') # appends the string 'two'
lst
```

```
[1, 'two']
```

```
lst.extend('three')
lst
```

```
[1, 'two', 't', 'h', 'r', 'e', 'e']
```

list.extend(item) expects item to have an iterator which is applied. Strings have iterators so each character in the string is added.

```
lst.extend(['four'])
lst
```

```
[1, 'two', 't', 'h', 'r', 'e', 'e', 'four']
```

list1 += list2 is equivalent to list1.extend(list2)



# list.insert(index, object)

```
lst:List[Any] = [1, 3]
```

```
lst.insert(1, 'two')  
lst
```

```
[1, 'two', 3]
```

```
lst.insert(99, 4)  
lst
```

```
[1, 'two', 3, 4]
```

`list.insert(index, object)` inserts object ***at a specific offset*** in the list

`list.insert(index,object)` appends the object if `index >= len(List)`



# Deleting items from a list

Delete an item(s) from a list with `del list[slice]`

```
lst:List[int] = [0, 1, 2, 3, 4]
del lst[0]  # delete the first element of lst
lst
```

```
[1, 2, 3, 4]
```

```
lst[2:]  # all but the first two elements
```

```
[3, 4]
```

```
del lst[2:]  # delete all but the last two elements
lst
```

```
[1, 2]
```

Note: this is the Python `del()` function, NOT `list.del()`  
`del()`, like `len()`, works on any Python sequence



# list.remove(value)

Remove an item **with a specific value** from a list

```
lst:List[int] = [0, 1, 2, 3, 4]
lst.remove(1)  # remove the first instance of 1
lst
```

```
[0, 2, 3, 4]
```

```
lst.remove(9)  # Raise a ValueError exception if not found
```

```
ValueError: list.remove(x): x not in list
```

Only the **first** instance is removed



## *list.count(value)*

Count the number of occurrences of *value* in a list

```
food = ['eggs', 'coffee', 'eggs', 'toast']
food.count('eggs')
```

2

```
food.count('donuts')
```

0



# list.index(value)

Return the index of the **first occurrence** of value in list

Raise **ValueError** if the value is not found

```
food:List[str] = ['eggs', 'coffee', 'eggs', 'toast']
food.index('eggs')
```

0

```
food.index('donuts')
```

**ValueError: 'donuts' is not in list**



# Value in list/Value *not* in list

Recall that we could say

```
'y' in 'Python'
```

True

‘in’ also works with values in a list

```
'eggs' in ['eggs', 'coffee', 'eggs', 'toast']
```

True

```
'donuts' in ['eggs', 'coffee', 'eggs', 'toast']
```

False



# list.pop(index)

Remove and return an item from a list at a specific index

list.pop(index) removes and returns the value at index

list.pop() removes and returns the last value in the list

del and remove remove the item, but don't return the value

lst.pop(n) where n > len(lst) raises IndexError

```
lst:List[int] = [1, 2, 3, 4, 5]  
lst.pop()
```

5

```
lst
```

```
[1, 2, 3, 4]
```

```
lst.pop(0)
```

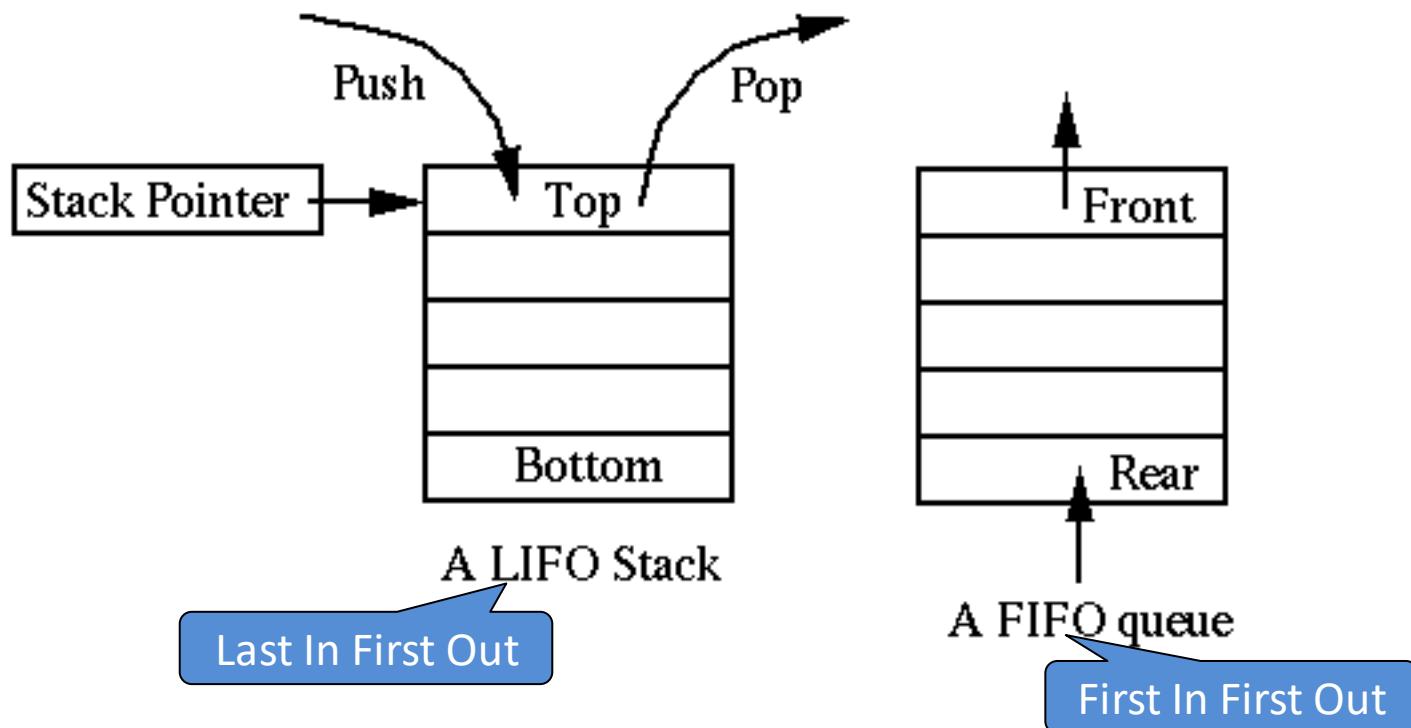
1

```
lst.pop(99)
```

IndexError: pop index out of range

# Stacks and Queues

Stacks and queues are very common data structures



Source: [http://alpcentauri.info/cpp\\_manual\\_stacks.htm](http://alpcentauri.info/cpp_manual_stacks.htm)

# Implement Stacks with a list

```
class Stack:
    def __init__(self) -> None:
        self.stack = list()

    def push(self, value:int) -> None:
        self.stack.insert(0, value)

    def pop(self) -> int:
        if len(self.stack) > 0:
            return self.stack.pop(0)
        else:
            raise IndexError("Attempted to pop from empty stack")
```

```
s = Stack()
s.push(1)
s.push(2)
s.pop()
```

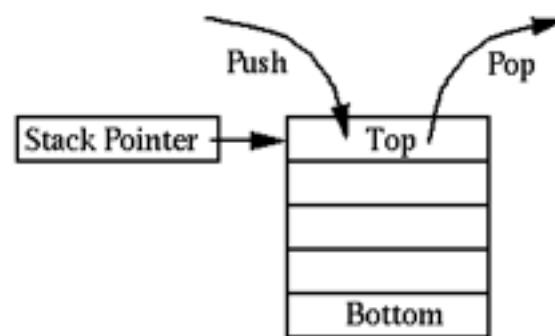
2 LIFO: Last In First Out

```
s.pop()
```

1 LIFO: Last In First Out

```
s.pop()
```

IndexError: Attempted to pop from empty stack



A LIFO Stack

# Implement Queues with a list

```
class Queue:
    def __init__(self) -> None:
        self.queue = list()

    def enqueue(self, value:int) -> None:
        self.queue.insert(0, value)

    def dequeue(self) -> int:
        if len(self.queue) > 0:
            return self.queue.pop()
        else:
            raise IndexError("Dequeue from empty queue")
```

```
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.dequeue()
```

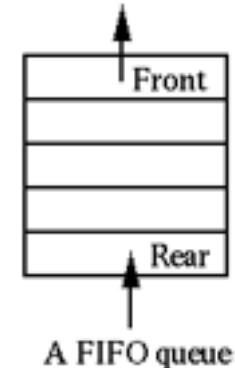
1 FIFO: First In First Out

```
q.dequeue()
```

2 FIFO: First In First Out

```
q.dequeue()
```

**IndexError: Dequeue from empty queue**



# Sorting lists

Two approaches to sort a list

1. `sorted(list)` – returns a sorted **copy** of the list
2. `list.sort()` - changes the list in place

```
lst1:List[int] = [3,1,2]  
sorted(lst1)
```

```
[1, 2, 3]
```

Python's `sorted()` function, not a method of class `list`

```
lst1
```

```
[3, 1, 2]
```

`sorted(lst1)` sorts a **copy** of `lst1` so `lst1` is unchanged

```
lst1.sort()
```

`list.sort()` is a statement, not an expression

```
lst1
```

```
[1, 2, 3]
```

`lst1` is changed because `lst1.sort()` sorts in place



# Sorting lists: descending

What if we want to sort a list in descending order?

Two approaches to sort a list

1. `sorted(list, reverse=True)` – returns a sorted **copy** of the list sorted from largest to smallest
2. `list.sort(reverse=True)` - changes the list in place

```
lst1:List[int] = [3,1,2]
sorted(lst1, reverse=True)

[3, 2, 1]
```

```
lst1.sort(reverse=True)
```

```
lst1
```

```
[3, 2, 1]
```

lst1 was changed in place



# List methods

Method	Action
list.append(element)	Append an element to a list
list.extend(sequence)	Concatenate list and sequence
list.insert(index, object)	Insert object at position index or at the end if index > len(list)
list.remove(value)	Remove the first instance of list with specified value or raise ValueError
list.count(value)	Count the number of instances of value in the list
list.index(value)	Return the index of value in list or raise ValueError if not found
list.pop(index)	Remove and return the value at index.
list.pop()	Remove and return the <b>last</b> value in the list
List.sort()	Destructively sort the elements in the list
sorted(list)	Sort a copy of the list
element in list	True if element is a member of list, else False



# Iterating through list elements

for loop iterates through list elements

```
for item in [1, 2, 3]:  
    print(item)
```

1  
2  
3

# Objects and references

```
x:str = "hello"  
y:str = "hello"
```

Strings are immutable so Python saves space by storing the string once and x and y reference the same object in memory



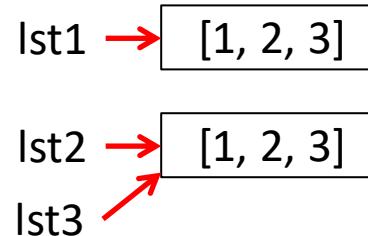
Python's `is` operator returns True if the same object

```
x is y
```

True

# Objects and references

```
lst1:List[int] = [1, 2, 3]
lst2:List[int] = [1, 2, 3]
lst3:List[int] = lst2
```



Lists are mutable so each list is stored separately

```
lst1 == lst2
```

True

```
lst1 is lst2
```

False

```
lst2 is lst3
```

True

lst1 and lst2 have the same values so `__eq__`

lst1 and lst2 are different instances of class list

lst2 and lst3 are the same instance of class list

# Objects and references

```
x:List[int] = [1, 2, 3]
y:List[int] = [1, 2, 3]
z:List[List[int]] = [x, y]
x == y
```

True

x **is** y

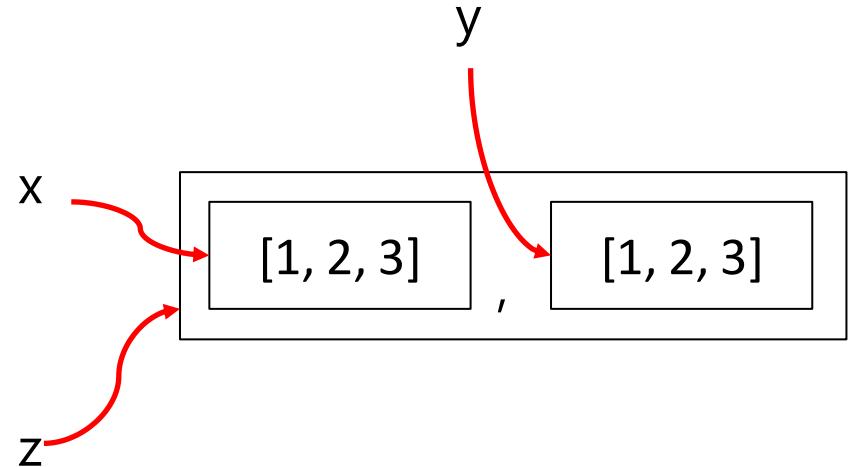
False

y **is** z[1]

True

x[0] = 4

What happens to z?



# Objects and references

```
x:List[int] = [1, 2, 3]
y:List[int] = [1, 2, 3]
z:List[List[int]] = [x, y]
x == y
```

True

```
x is y
```

False

```
y is z[1]
```

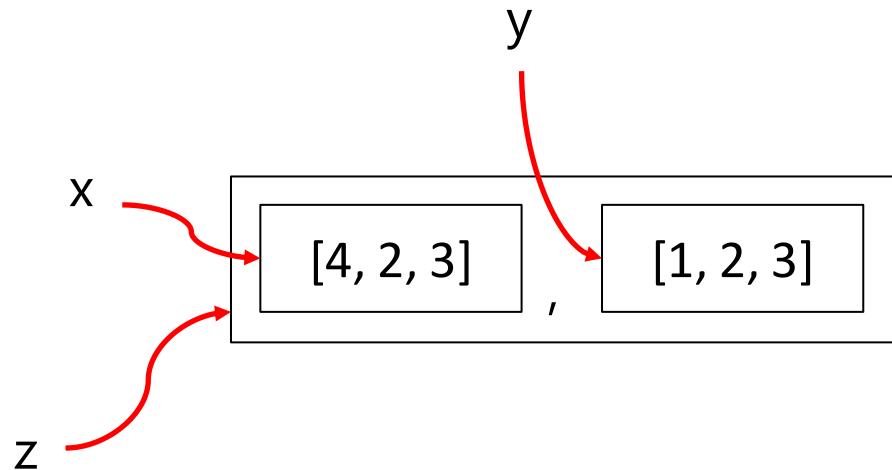
True

```
x[0] = 4
```

What happens to z?

```
z
```

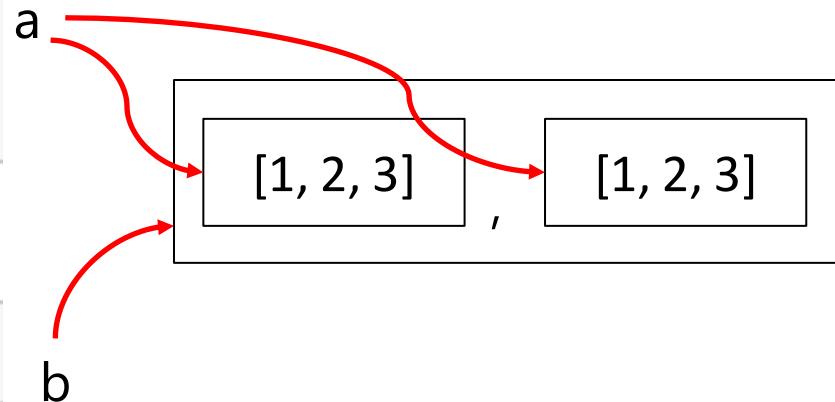
```
[[4, 2, 3], [1, 2, 3]]
```



# Objects and references

```
a:List[int] = [1, 2, 3]
b:List[List[int]] = [a, a]
b
[[1, 2, 3], [1, 2, 3]]
```

```
b[1][1] = 4
```



What happens to `b`?  
What happens to `a`?

# Objects and references

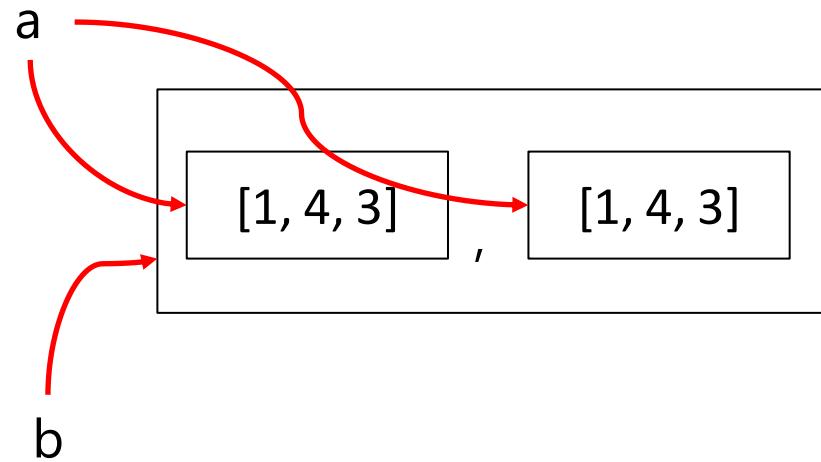
```
a:List[int] = [1, 2, 3]
b:List[List[int]] = [a, a]
b
```

```
[[1, 2, 3], [1, 2, 3]]
```

```
b[1][1] = 4
```

```
b
```

```
[[1, 4, 3], [1, 4, 3]]
```



Just be aware so you're not surprised



# Careful with sorting in place

```
lst1:List[int] = [3, 1, 2]  
lst2:List[Any] = [lst1, 4]  
lst2
```

```
[[3, 1, 2], 4]
```

```
lst1.sort()
```

```
lst1
```

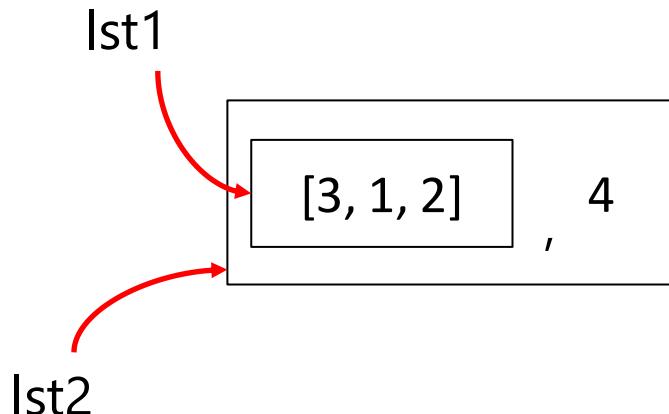
```
[1, 2, 3]
```

lst1 was changed in place

```
lst2
```

```
[[1, 2, 3], 4]
```

lst2 was also changed



Choose `list.sort()` and `sorted(list)` carefully!

Both are appropriate in different circumstances



# Passing lists as function arguments

Python passes arguments by assignment

As a "pointer" to the object rather than a copy

Lists passed to functions as parameters may be changed  
by the function as a side effect

```
def my_append(item:int, lst:List[int]) -> None:  
    lst.append(item)  
  
lst1:List[int] = [1, 2, 3]  
my_append(4, lst1)  
lst1
```

```
[1, 2, 3, 4]
```

Just be aware so you're not surprised  
Keep this in mind when debugging!



# CAUTION!!!

## Never iterate over a changing a list

Iterating over a copy works properly

```
def remove_th(s:str) -> str:  
    """ Given string s, split s into words then remove words  
        beginning with 'th'. Then join the words back together.  
    """  
    words:List[str] = s.split()  
    for word in words:  
        print(f"checking: {word}")  
        if word.lower().startswith('th'):  
            words.remove(word)  
    return " ".join(words)
```

```
remove_th("this thing that is the problem")
```

```
checking: this
```

```
checking: that
```

```
checking: the
```

```
'thing is problem'
```

Iterating over a changing list is  
dangerous

checking 'this': OK

skipped checking 'thing'

skipped checking 'is'

skipped checking 'problem'



# CAUTION!!!

## Never iterate over a changing a list

```
def remove_th(s:str) -> str:  
    """ Given string s, split s into words then remove words  
        beginning with 'th'. Then join the words back together.  
    """  
  
    words:List[str] = s.split()  
    for word in words.copy():  
        print(f"checking: {word}")  
        if word.lower().startswith('th'):  
            words.remove(word)  
    return " ".join(words)  
  
remove_th("this thing that is the problem")
```

checking: this  
checking: thing  
checking: that  
checking: is  
checking: the  
checking: problem  
  
'is problem'

Iterate over a **copy** of the list for the expected behavior

Modify the **original** list

Here's the words to check

That's better!



# A more "Pythonic" solution

```
def remove_th(s:str) -> str:  
    """ Given string s, split s into words then remove words  
        beginning with 'th'. Then join the words back together.  
    """  
  
    keep:List[str] = list() ————— A list of words to keep  
    words:List[str] = s.split()  
    for word in words:  
        if not word.lower().startswith('th'):  
            keep.append(word) ————— Add this word to keep  
    return " ".join(keep)  
  
remove_th("this thing that is the problem")
```

'is problem' ————— That's better!

We'll see how to use a list comprehension to simplify the function



# Using lists

Write a function that given a list, returns a list of distinct values  
Use the same approach as `remove_th()`

This is a common pattern in Python programs

```
def distinct_items(l>List[int]) -> List[int]:  
    """ Given a list, return a new list the distinct values """  
    distinct:List[int] = list() # items we've seen so far  
  
    for item in l:  
        if item not in distinct:  
            distinct.append(item)  
  
    return distinct  
  
distinct_items([3,3,1,2,1,2,4,3])
```

The values we want to keep

Keep this value

[3, 1, 2, 4]



# List comprehensions

Transforming lists is very common in Python programs

List comprehensions are a syntactic shortcut

```
result:List[Any] = list() # values to keep
for item in [1, 2, 3]:
    result.append(item + 1)

result
```

[2, 3, 4]

Same logic with list comprehensions (1 line vs. 3 lines)

```
[ item + 1 for item in [1 ,2, 3] ]
```

[2, 3, 4]



# Conditional list filtering

```
result:List[Any] = list() # values to keep
for item in [1, 2, 3, 4]:
    if item % 2 == 0:
        result.append(item)
result
```

[2, 4]

Same logic with list comprehensions (1 line vs. 4 lines)

```
[ item for item in [1, 2, 3, 4] if item % 2 == 0 ]
```

[2, 4]



# remove\_th() with list comprehensions

```
def remove_th(s:str) -> str:  
    words:List[str] = s.split()  
    keep:List[str] = [word for word in words if not word.startswith('th')]  
    return " ".join(keep)  
  
remove_th("this thing that is the problem")  
  
'is problem'
```



# distinct\_values() with list comprehensions?

```
def distinct_items(l>List[int]) -> List[int]:
    """ Given a list, return a new list the distinct values """
    distinct:List[int] = list() # items we've seen so far

    for item in l:
        if item not in distinct:
            distinct.append(item)

    return distinct

distinct_items([3,3,1,2,1,2,4,3])
```

```
[3, 1, 2, 4]
```

List comprehensions don't help in this context because we need to access the intermediate list of distinct values

# Nested list comprehensions

```
def flatten(l>List[List[Any]]) -> List[Any]:
    """ Given a list of lists,
        flatten the elements into a single list
    """
    result>List[Any] = list()
    for sublist in l:
        for item in sublist:
            result.append(item)
    return result

flatten([[1, 2], [3, 4], [5, 6]])
```

[1, 2, 3, 4, 5, 6]

```
def flatten2(l>List[Any]) -> List[Any]:
    """ flatten with list comprehensions """
    return [ item for sublist in l for item in sublist ]
```

flatten2([[1, 2], [3, 4], [5, 6]])

[1, 2, 3, 4, 5, 6]



# Debugging list comprehensions

Debugging list comprehensions can be challenging

1. Try reading the code
2. Expand the code and debug
3. Rewrite as a list comprehension

The diagram illustrates the equivalence between a list comprehension and its expanded form. At the top, a list comprehension `[ item + 1 for item in [1, 2, 3] ]` is shown. A green box highlights the expression `item + 1`, and a red box highlights the loop part `for item in [1, 2, 3]`. Below this, the resulting list `[2, 3, 4]` is shown. In the middle, the expanded code is shown in a grey box:

```
result:List[Any] = list() # values to keep
for item in [1, 2, 3]:
    result.append(item + 1)

result
```

A red arrow points from the red box in the list comprehension to the `for` loop in the expanded code. A green arrow points from the green box in the list comprehension to the expression `item + 1` in the expanded code. At the bottom, the resulting list `[2, 3, 4]` is shown again.



# Optimizing algorithms

Sometimes, a little extra thought can dramatically improve the performance of our algorithms

Sometimes, storing extra information can help...

***Write a generator to compute all prime numbers***

How would you do it?





# Naïve prime number generator

```
from typing import Iterator

def next_prime() -> Iterator[int]:
    """ generate a stream of prime numbers """
    cur:int = 2 # 2 is the first prime by definition
    while True:
        for n in range(2, cur):
            if cur % n == 0:
                break # n is a divisor of cur so n is not prime
        else:
            yield cur # didn't divide evenly by any number so prime
        cur += 1 # check the next value
```

How many times do we execute this statement?

for/else simplifies code in some situations

```
primes:Iterator[int] = next_prime()
[next(primes) for n in range(15)]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

# Prime number generation

**Task:** Write a generator to compute all prime numbers  
How many modulo calculations? How long?

N	Mod Calculations	Time (seconds)
100	1233	0.08
200	4426	0.39
400	14,845	1.66
800	51,488	7.50

Can we reduce the number of modulo calculations?



# Better prime number generation

Can we reduce the number of modulo calculations?

Yes! Divide only by prime numbers  $\leq n$

But then we need to store all prime numbers  $\leq n$

**Speed/Space trade off**





# Better prime number generation

Change generator to store all primes as they are calculated!

```
def next_prime() -> Iterator[int]:
    """ generate prime numbers but store the primes as discovered
        and compare only against the list of primes.
    """
    primes:List[int] = list() # store all the primes we found so far
    cur:int = 2 # cur is the next number to be checked

    while True:
        for p in primes:
            if cur % p == 0:
                break
        else:
            # exhausted all of the primes - found another prime
            primes.append(cur)
            yield cur

        cur += 1

primes:Iterator[int] = next_prime()
[next(primes) for n in range(15)]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```



# Better prime number generation

How many comparisons? How long?

N	Naïve Mod Calculations	Improved Mod Calculations	Naïve Time (seconds)	Improved Time (seconds)
100	1233	438	0.08	0.04
200	4426	1283	0.39	0.08
400	14,845	3642	1.66	0.28
800	51,488	10,830	7.50	1.03

A small change to the algorithm makes  
a big performance improvement!





# Optional Homework 06: reorder

Python provides efficient sort utilities for all sequences

Useful to understand a few simple sorting algorithms

Insertion sort is probably the simplest to implement

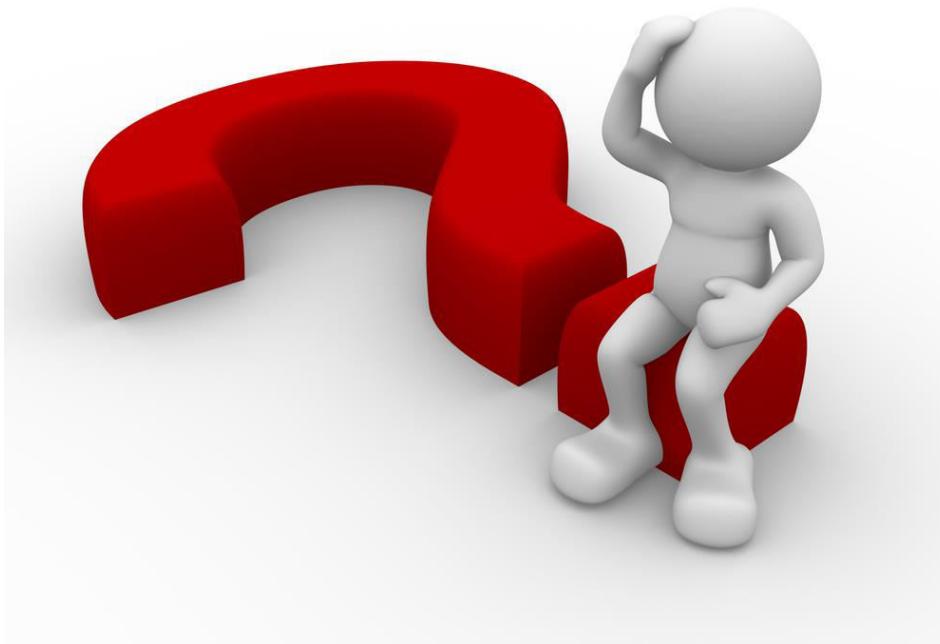
Simply insert elements into a new list in the proper spot

reorder(5, 1, 3, 2)

Result before insertion	New item to insert	Result after insertion
[ ]	5	[5]
[5]	1	[1, 5]
[1, 5]	3	[1, 3, 5]
[1, 3, 5]	2	[1, 2, 3, 5]

You MUST use this approach for HW06, not Google!

# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Containers: Tuples, Dictionaries and Sets*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

And

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

<https://docs.python.org/3/library/stdtypes.html>

# Today's topics

Lambda Expressions

Containers

Lists

Tuples

Dictionaries

Counter

DefaultDict

Sets

Frozensets





# lambda: anonymous functions

lambda expressions are one line, anonymous functions

lambda *parameters*: *expression*

Is equivalent to

```
def <anon>(parameters): return expression
```

*expression* must be a **pure** expression

**No** assignment, **if**, **while**, **for**, **try**, **return**, ... statements

Lambda expressions are fairly **uncommon** in Python 3.\*

lambda was popular before list comprehensions

Used frequently with `map()`, `filter()`, and `reduce()`

Popular when using Python as a Functional language



# When to use `lambda`?

lambda expressions are commonly used to change the default behavior of `sort`

Sort names by the last character in the name

key must be a function

```
names: List[str] = ['Sujit', 'Yujie', 'Fred', 'Nanda']
sorted(names, key=lambda n: n[-1])
['Nanda', 'Fred', 'Yujie', 'Sujit']
```

Simpler than  
`def last_char(s):  
 return s[-1]`



# When to use `lambda`?

lambda expressions are commonly used to define the factory function in a defaultdict

```
student: Dict[str, Dict[str, int]]  
student = defaultdict(lambda: defaultdict(int))  
student["Gaijie"]["Credits"] += 3  
student["Gaijie"]["Credits"]
```

3

The factory must be a function so lambda is a good solution



# Containers

Python provides several different types of containers

**Lists** – ordered, arbitrary values and mutable – [1, 2, 3]

**Tuples** - like list, but immutable – (1, 2, 3) or 1, 2, 3

**Dictionaries** – unordered Key/Value pairs

– {'one':1, 'two':2, 'three':3 }

**Sets** - {1, 2, 3 }

- Like lists, but with only unique values
- Like lists, but unordered
- Like dictionaries with keys, but no values

Python 3.7+  
guarantees dict  
ordered by  
insertion of keys

These containers are fundamental in Python and are used in most Python programs



# Indices and slices with tuples

Indices and slices work the same with tuples as with strings and lists

```
from typing import Tuple
```

Note: parens, not brackets

```
t: Tuple[str, str, str] = ('eggs', 'coffee', 'toast')  
t[0:2]
```

```
('eggs', 'coffee')
```

```
t[0] = 'donuts'
```

Tuples can't be changed

```
TypeError: 'tuple' object does not support item assignment
```



# Tuple methods

Many of the list methods also work with tuples

```
t2: Tuple[str, str, str]  
t2 = 'eggs', 'coffee', 'toast'
```

Parens are optional

```
t2.index('toast')
```

2

Tuple.index(value) and  
tuple.count(value) work just like lists

```
t2.count('toast')
```

1

None of the list methods that add, change, or delete elements are defined for tuples because tuples are immutable



# Tuple unpacking

Tuples enable multiple assignments in a single statement

```
v1: str  
v2: int  
v3: float  
v1, v2, v3 = 'a', 2, 3.14  
print(v1, v2, v3)
```

a 2 3.14



# Value swapping with tuple unpacking

```
v1: int
v2: int
v1, v2 = 1, 2

# C++/Java style value swapping
tmp: int = v1
v1 = v2
v2 = tmp

v1, v2 = 3, 4 # reinitialize

# swapping with tuple unpacking
v1, v2 = v2, v1 # swap values in one statement
print(v1, v2)
```

4 3



# Returning multiple values with tuple unpacking

Tuples are ideal for returning multiple values from a function

```
def next3(n: int) -> Tuple[int, int, int]:  
    return n + 1, n + 2, n + 3
```

```
a, b, c = next3(1)  
print(a, b, c)
```

2 3 4



# Assigning multiple variables with tuple unpacking

Tuple unpacking improves readability when assigning multiple variables

```
values: List[Tuple[str, str]] = [('1.1', '1.2'), ('2.1', '2.2')]

for item in values:
    print(item[0], item[1]) # okay
```

Syntactically correct but hard to read

```
1.1 1.2
2.1 2.2
```

Easier to read and understand

```
for first, second in values: # better names
    print(first, second)
```

```
1.1 1.2
2.1 2.2
```



# Iterating through tuple elements

for loop iterates through tuple elements

```
for elem in 1, 2, 3:  
    print(elem)
```

Comma specifies a tuple

1  
2  
3

# When/why to use tuples

Tuple unpacking is great for...

- Assigning multiple values

- Returning multiple values from a function

- Swapping values

Tuples are more efficient than lists

Tuples protect your data from unintentional changes

Tuples can be used as keys in dicts





# Dictionaries

Many applications require key/value pairs

How many times does a word occur?

Mapping from one value to another

“one” → “uno”, “two” → “dos”, ...

Book index (on which pages does a word occur?)

“Python” → [1, 4, 17, 19, 20]

Dictionaries are a perfect solution for these tasks!

Dictionaries map **keys** to **values**

Dictionaries provide associative arrays/hash tables

# Finding an item in a collection

I need to find my friend Nanda

I can knock on every door until  
I find him ...

... or I can look up his address  
and go directly to the right  
door

Dictionaries map keys to values

Nanda → Apt 5B





# Create a dictionary

```
from typing import Dict

d1: Dict[str, int] = dict() # create an empty dict
d2: Dict[str, int] = {} # also creates an empty dict
d3: Dict[str, int] = {'one': 1, 'two': 2, 'three': 3}
d3

{'one': 1, 'two': 2, 'three': 3}
```

Dictionaries are stored in order the keys are added as of Python 3.7

```
values4: List[Tuple[str, int]]
values4 = [('three', 3), ('two', 2), ('one', 1)]

d4: Dict[str, int] = dict(values4)

d3 == d4
```

True

Initializing a dictionary requires pairs of keys and values



# dict

**Keys** can be any **hashable** value:

`str, int, float, tuple, frozenset`

**Cannot** use a `list` or `set` as a dictionary key

Lists and sets may change so the hash value could change

**Values** can be any type, even dictionaries

Perfect for database types with rows and columns

Keys and values may be heterogeneous

Accessed by **value**, not **position** (like strings, lists, tuples)

Fast, random access to values based on keys



# Accessing and setting values

```
d: Dict[str, int] = {'apples': 3, 'pears': 10, 'peaches': 12}  
d['bananas'] = 5
```

Add new key 'bananas' with value 5

```
d['frogs']
```

```
KeyError: 'frogs'
```

Access value of unknown key 'frogs' so raise KeyError

```
d.get('peaches')
```

dict.get(key) gets the value associated with 'key', but  
does not add 'key' to the dict

```
12
```

'frogs' is still not a key, but no exception

```
d.get('frogs') is None
```

```
True
```

```
d.get('frogs', 22)
```

If 'frogs' is not a key, then return 22.  
'frogs' is NOT added as a key

```
22
```



# setdefault(key, value=None)

*Set `dict[key] = value` if `key` not in `dict`*

*Set the value for a **specific** key if the value is not already set*

```
d: Dict[str, int] = {'apples': 3, 'pears': 10, 'peaches': 12}
```

```
d.setdefault('plums')  
d['plums'] is None
```

‘plums’ is not in d so `d['plums'] = None`

True

```
d.setdefault('grapes', 15)
```

‘grapes’ is not in d so `d['grapes'] = 15`

15

```
d['grapes']
```

15

```
d.setdefault('grapes', 22)
```

15

```
d['grapes']
```

‘grapes’ is in d so don’t change `d['grapes']`

15



# Replace long if/elif/else blocks with dict



86°F = 30°C  
77°F = 25°C  
68°F = 20°C  
59°F = 15°C  
50°F = 10°C  
41°F = 5°C  
32°F = 0°C  
23°F = -5°C  
14°F = -10°C

```
def f2c_6train(f: int) -> Optional[int]:  
    c: Optional[int] = None  
    if f == 86:  
        c = 30  
    elif f == 77:  
        c = 25  
    elif f == 68:  
        c = 20  
    elif f == 59:  
        c = 15  
    elif f == 50:  
        c = 10  
    elif f == 41:  
        c = 5  
  
    return c
```

Replace this long,  
error prone if  
statement with a  
dict

```
def f2c_6train(f: int) -> int:  
    f2c_map: Dict[int, int] = \  
        {86: 30, 77: 25, 68: 20, 59: 15, 50: 10, 41: 5}  
    return f2c_map.get(f)
```



# Iterating over a dictionary

```
d: Dict[str, int] = {'apples': 3, 'pears': 10, 'peaches': 12}  
for key in d.keys():  
    print(key)
```

apples  
pears  
peaches

for key in d:  
is equivalent to  
for key in d.keys():

```
for value in d.values():  
    print(value)
```

3  
10  
12

```
for key, value in d.items():  
    print(key, value)
```

apples 3  
pears 10  
peaches 12

dict.items() returns a sequence of (key, value)  
dict.keys() returns a sequence of keys  
dict.values() returns a sequence of values



# Finding dict's max key or key associated with the max value

How can we find the max/min key in a dictionary?

```
d: Dict[str, int] = {'a': 10, 'b': 15, 'c': 7}  
d
```

```
{'a': 10, 'b': 15, 'c': 7}
```

```
max(d)      Return the max key value
```

```
'c'
```

```
max(d, key=d.get)
```

```
'b'
```

```
d[max(d, key=d.get)]
```

Return the key with the max value:  
`max(dict, key=dict.get)` returns the key  
associated with the max value in `dict`



# Implementing tables with nested dicts

CWID	FName	LName
CWID00	Jin	He
CWID01	Nanda	Koka
CWID02	Sally	Fields

```
students: Dict[str, Dict[str, str]] = {  
    'CWID00': {'FName': 'Jin', 'LName': 'He'},  
    'CWID01': {'FName': 'Nanda', 'LName': 'Koka'},  
    'CWID02': {'FName': 'Maha', 'LName': 'Aldrisi'},  
}  
  
students['CWID01']
```

```
{'FName': 'Nanda', 'LName': 'Koka'}
```

```
students['CWID01']['LName']
```

```
'Koka'
```



# Use dicts to count items: Attempt I

Calculate the frequency of each item in a sequence

```
freq: Dict[str, int] = dict()  
for c in "Hello world":  
    freq[c] += 1
```

**KeyError:** 'H'

'H' is not a key in freq, so raise exception



# Use dicts to count items: Attempt 2

Calculate the frequency of items in a sequence

```
freq: Dict[str, int] = dict()  
for c in "Hello world":  
    if c in freq:  
        freq[c] += 1 # subsequent occurrence  
    else:  
        freq[c] = 1 # first occurrence
```

---

This solution is correct, but we can do better



# Use dicts to count items: Success!

Calculate the frequency of each character

```
freq: Dict[str, int] = dict()  
for c in "Hello world":  
    freq[c] = freq.get(c, 0) + 1
```

If `c` is already a key,  
then get the value  
else use 0.  
Then Increment the value

This solution is correct, but we can do better



# Special case dictionaries: Counter

Counting occurrences of some item is a common task  
e.g. count characters, words, list elements, etc.

```
from collections import Counter
import typing
```

```
c: typing.Counter[str] = Counter("Hello world")
c
```

```
Counter({'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

```
c.most_common(2)
```

Return the N most frequent items

```
[('l', 3), ('o', 2)]
```

```
list(c.elements())
```

Return an iterator of all of the elements in undefined order

```
['H', 'e', 'l', 'l', 'l', 'o', 'o', ' ', 'w', 'r', 'd']
```



# Special case dictionaries: defaultdict

`collections.defaultdict(factory)` assigns a default value from the specified factory if the key is not found

```
from collections import defaultdict
from typing import DefaultDict
dd:DefaultDict[str, int] = defaultdict(int)
int()
```

Int() returns 0 which gives a default of 0

```
dd['new key']  
0
```

The default value of any new key is 0

```
# option 1: dict.get(key, default)
freq: Dict[str, int] = dict()
for c in "Hello world":
    freq[c] = freq.get(c, 0) + 1
```

These two options do the same thing.  
Which is easier to read?

```
# option 2: defaultdict(int)
freq: DefaultDict[str, int] = defaultdict(int)
for c in "Hello world":
    freq[c] += 1
```

If c is already a key, then return freq[c]  
else freq[c] = 0, then increment the value



# defaultdict(*factory*)

You can define a defaultdict with any type

```
from collections import defaultdict
from typing import DefaultDict
translate:DefaultDict[int, str] = defaultdict(str)

translate[1] = 'uno'
```

```
translate[2]
```

```
''
```

If the key does not appear in the dict, then add the key to the dict and return the default value specified by the factory

```
ingredients: DefaultDict[str, List[str]] = defaultdict(list)
ingredients['bread'] = ['flour', 'water']
ingredients['soup'].append('milk')
ingredients['soup']

['milk']
```

Default value is an empty list

Safe to append because value is a list by default, even if key hasn't been added



# defaultdict(factory)

You can define a defaultdict with classes

```
class Student:  
    def __init__(self) -> None:  
        self.classes: List[str] = list()  
  
    def add_class(self, cls_name: str) -> None:  
        self.classes.append(cls_name)  
  
students: DefaultDict[str, Student] = defaultdict(Student)  
  
students['123456'].add_class('SSW 810')
```

The class `__init__(self)`  
must take no additional  
arguments



# defaultdict(factory)

## How to specify a different default value?

```
from collections import defaultdict
from typing import DefaultDict
default0: DefaultDict[int, int] = defaultdict(int)
default0[3]
```

0

lambda defines an anonymous function

```
default42: DefaultDict[int, int] = defaultdict(lambda: 42)
default42[3]
```

42

The default value is integer 42

```
default_blue: DefaultDict[int, str] = defaultdict(lambda: "blue")
default_blue[3]
```

'blue'

The default value is string 'blue'



# Defaultdict with multiple keys

What if we want a `defaultdict` with multiple keys?

```
from collections import defaultdict
from typing import DefaultDict

rps_outcome: DefaultDict[str, str] = \
    defaultdict(lambda: defaultdict(lambda: "It's a tie"))

rps_outcome['rock']['paper'] = "Rock covers paper!"
rps_outcome['scissors']['paper'] = "Scissors cut paper!"

rps_outcome['rock']['rock']

"It's a tie"
```

`rps_outcome[human]` is a `defaultdict(lambda: "It's a tie!")`

This provides a default outcome for unspecified combinations of human and computer moves

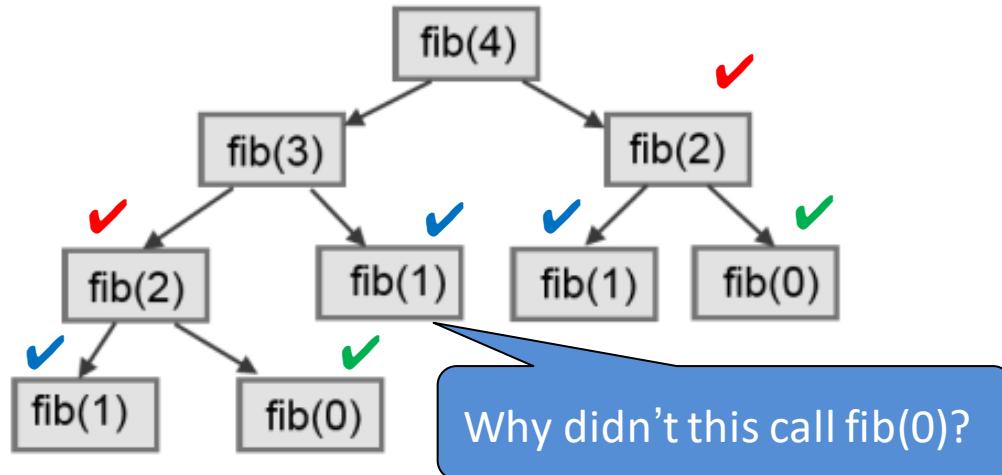
# Memoization/caching

Memoization speeds up calculations by creating and maintaining a cache of previous results

Don't recalculate each time, just retrieve it from the cache

Dictionaries are a great way to implement caches

Examples: recursive factorial, recursive Fibonacci



Call	Count
fib(4)	1
fib(3)	1
fib(2)	2
fib(1)	3
fib(0)	2



# Memoizing Fibonacci: Performance

```
def fib(n: int) -> int:  
    """ return the nth number in the fibonacci sequence """  
    if n in (0, 1):  
        return n  
    else:  
        return fib(n - 1) + fib(n - 2)  
  
fib_cache: Dict[int, int] = {0: 0, 1: 1}  
  
def fibc(n: int) -> int:  
    if n not in fib_cache:  
        fib_cache[n] = fibc(n - 1) + fibc(n - 2)  
  
    return fib_cache[n]
```

Populate the cache

The cache solution is > 2 million times faster for fib(40)

Approach	Calls	Time (Seconds)
fib(40)	331M	74.59
fibc(40)	78	0.00033

# Sets

A set is like a dictionary with **keys**, but **no values**

Each value can appear only once

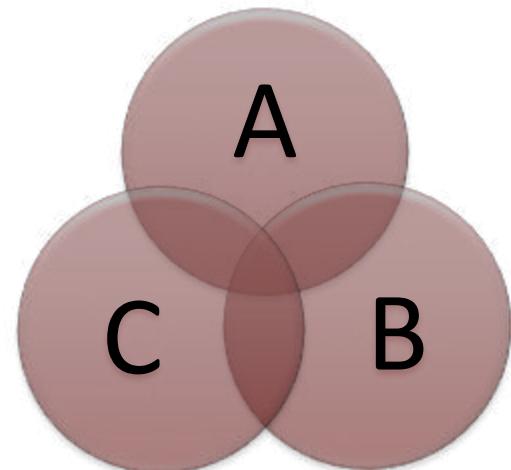
Values are unordered

Values can be any **hashable** type (int, float, string, tuple, ...)

Sets **cannot** contain lists or sets because they are **mutable**

Set values can be heterogeneous

Sets include methods for set operations





# Set methods

```
from typing import Set, Any
s: Set[Any] = set([1, 'two'])
s.add(3)
s
{1, 3, 'two'}
```

Note the order is undefined

```
len(s)
```

```
3
```

```
3 in s
```

```
True
```

```
s.add(3)
```

Wait!!! 3 is already a member of the set s!  
Adding an element that is already a member of the set is a no-op. Adding a duplicate does *not* raise an exception

```
s
```

```
{1, 3, 'two'}
```



# Set methods from high school math

```
s1: Set[Any] = set([1, 'two', 3])  
s2: Set[Any] = {1, 'two', 3, 4, 'five'}  
s1 <= s2
```

True

Is s1 a subset of s2?

Create a set from  
a sequence

```
s3: Set[Any] = {'two', 3, 'six'}  
s1.intersection(s3)
```

{3, 'two'}

s1  $\cap$  s3

```
s1.union(s3)
```

s1  $\cup$  s3

{1, 3, 'six', 'two'}

Use *curly brackets*  
to define a new set,  
like a dictionary  
without keys



# Set methods from high school math

```
s1: Set[int] = {1, 2, 3}
```

```
s2: Set[int] = {2, 3, 4}
```

```
s2.difference(s1)
```

```
{4}
```

Elements in s2, but not in s1

```
s1.symmetric_difference(s2)
```

```
{1, 4}
```

Elements in s1 or s2, but not both



# Finding unique items

Sets provide a convenient way to identify the distinct values from an iterator, e.g. string, list, tuple

```
set('hello')
```

```
{'e', 'h', 'l', 'o'}
```

```
set([3, 1, 4, 1, 5])
```

```
{1, 3, 4, 5}
```

How many distinct characters in “Mississippi”?

```
len(set('Mississippi'))
```

4



# Write a program...

Given a sequence,  $s$ , return a list of lists where each inner list is a pair of the unique values in  $s$  and the number of occurrences of each value.

E.g. `counts('Mississippi')` returns `[('M', 1), ('i', 4), ('s', 4), ('p', 2)]`

```
def counts(s: str) -> List[Tuple[str, int]]:  
    return [(item, s.count(item)) for item in set(s)]
```

```
counts("Mississippi")
```

```
[('i', 4), ('s', 4), ('p', 2), ('M', 1)]
```

**Beware:** Sets are unordered so you may need to sort the result to compare in test cases



# frozense

A `frozense` is an immutable set

The same operations as a set except the items in the set can't change

- Can't add new items to the set

- Can't delete items from the set

`frozense` is hashable so it **can** be used as a dictionary key



# frozenset as a dictionary key

```
d: Dict[set, int] = dict()  
s: Set[int] = {1, 2}  
s.add(3)  
d[s] = 4
```

s is a set and is mutable

**TypeError:** unhashable type: 'set'

Dictionary keys must be  
hashable items

```
fs: FrozenSet[int] = frozenset(s)  
fs.add(4)
```

frozensets are not mutable

**AttributeError:** 'frozenset' object has no attribute 'add'

```
d[fs] = 4
```

frozensets are hashable and can be  
used as dictionary keys

```
d
```

```
{frozenset({1, 2, 3}): 4}
```

# When to use which container?

## Lists

Ordered items of any type

Duplicates are fine

Add to or delete from beginning or end

Great for stacks (LIFO) and queues (FIFO)

## Tuples

Immutable version of lists

Grouping multiple related values as a single item

Returning multiple values from a function

Tuple unpacking for multiple assignments or swapping values

Lightweight classes without methods



# When to use which container?

## Dictionary

Key/Value pairs

Good when associating information with a key

Great for look up tables

Great for counting distinct elements

Replacing long if/elif/else blocks

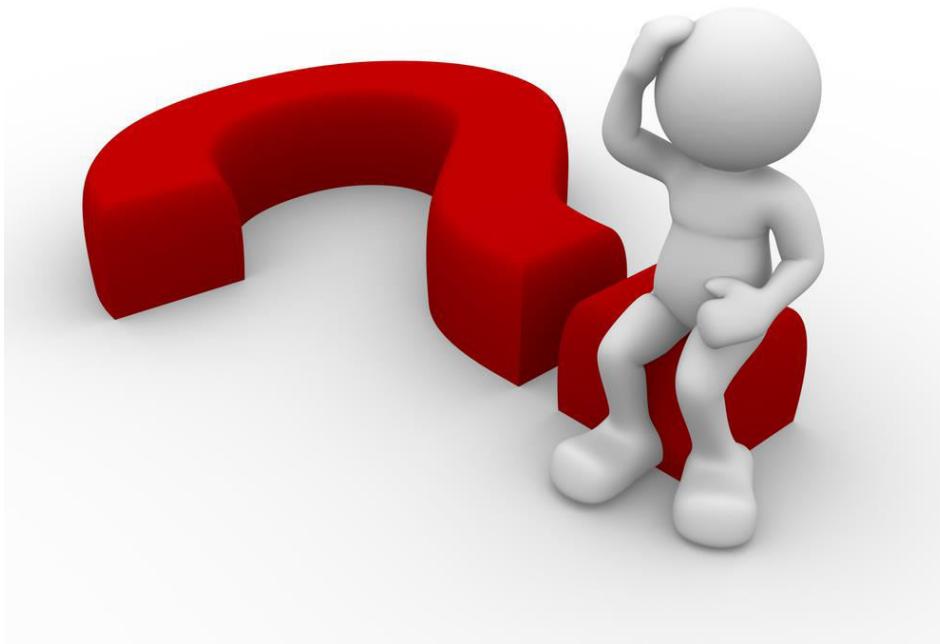


## Sets

Great any time you need set operations

Easy to identify distinct elements from an iterator, e.g. string, list, tuple

# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Advanced Function Parameters Python Modules*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from:

“How to Think Like a Computer Scientist: Learning with Python 3 (RLE)”, Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers,

<http://openbookproject.net/thinkcs/python/english3e/>

“Introducing Python”, Bill Lubanovic, ISBN-13: 978-1449359362

<http://proquest.safaribooksonline.com/book/programming/python/9781449361167>

<https://docs.python.org/3/library/stdtypes.html>

<https://mkaz.tech/python-string-format.html>

# Today's topics

Advanced function parameters

Python modules and packages

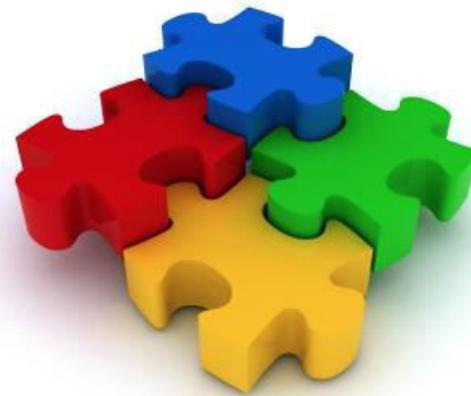
sys

time

datetime

os

prettytable





# Function parameters and arguments

A function or method defines **parameters**

**Arguments** are the values that are passed to the function or method

Parameters are assigned to arguments by position

**Parameters** are used in the function/method definition

```
def myfunc(param1: int, param2: int, param3: int) -> None:  
    print(param1, param2, param3)
```

```
myfunc(1, 2, 3)
```

```
1 2 3
```

**Arguments** are passed to the function and are assigned to the parameters by position (tuple unpacking)



# Function parameters by *name*

Arguments may also be passed by *name*

```
def myfunc(param1: int, param2: int, param3: int) -> None:  
    print(param1, param2, param3)
```

```
myfunc(param3=6, param1=4, param2=5)
```

4 5 6

```
myfunc(param2=5, param3=6, param1=4)
```

4 5 6

Arguments may be specified  
in any order when passing  
by **name** (*well, almost...*)



# Mix positional and named arguments

All positional arguments must be specified before any named arguments

```
def myfunc(param1: int, param2: int, param3: int) -> None:  
    print(param1, param2, param3)
```

```
myfunc(1, param3=3, param2=2)
```

```
1 2 3
```

```
myfunc(param3=3, 1, param2=2)
```

```
File "<ipython-input-5-3c54a65effa1>", line 1  
    myfunc(param3=3, 1, param2=2)  
          ^
```

```
SyntaxError: positional argument follows keyword argument
```

Mix positional and named arguments

All positional arguments must precede name arguments, i.e. named parameters must be specified last



# Optional parameters with default values

Functions/methods may define optional parameters that assume default values if corresponding arguments are not passed by the function call

```
def myfunc(param1: int, param2: int = 2, param3: int = 3) -> None:  
    print(param1, param2, param3)
```

```
myfunc(1, 2, 3)
```

```
1 2 3
```

```
myfunc(1, 2)
```

```
1 2 3
```

```
myfunc(1)
```

```
1 2 3
```

```
myfunc()
```

```
TypeError: myfunc() missing 1 required positional argument: 'param1'
```

param1 is required  
param2 defaults to 2 if not specified  
param3 defaults to 3 if not specified

Param1 is required: no default value



# Mixing optional and named parameters

Function calls may mix positional, named, and optional arguments

```
def myfunc(param1: int, param2: int = 2, param3: int = 3) -> None:  
    print(param1, param2, param3)  
  
myfunc(4, param3=6)
```

4 2 6

param1 == 4  
param2 defaults to 2  
param3 passed by name



# BEWARE!!!

Mutable default parameters may have unexpected surprises

```
def watchout(value: int, lst: List[int] = []) -> None:  
    print(f"before: lst={lst}")  
    lst.append(value)  
    print(f"after: lst={lst}")
```

```
watchout(3, [1, 2])
```

```
before: lst=[1, 2]  
after: lst=[1, 2, 3]
```

So far, so good

```
watchout(4)
```

```
before: lst=[]  
after: lst=[4]
```

That's right... What's the problem?

```
watchout(5)
```

```
before: lst=[4]  
after: lst=[4, 5]
```

WAIT!!! That should be [5]

Python defines the default value lst when the function is defined

Each subsequent call using the default value mutates the value from the definition

# What happened?!?!

Mutable default parameters may have unexpected surprises

```
def watchout(value: int, lst: List[int] = []) -> None:  
    print(f"before: lst={lst}")  
    lst.append(value)  
    print(f"after: lst={lst}")
```

```
watchout(3, [1, 2])
```

```
before: lst=[1, 2]  
after: lst=[1, 2, 3]
```

lst uses the value passed into the function, i.e.  
the default value was not used

```
watchout(4)
```

```
before: lst=[]  
after: lst=[4]
```

lst has the default value from when the  
function was defined, then append 4

```
watchout(5)
```

```
before: lst=[4]  
after: lst=[4, 5]
```

lst == [4] before appending 5



# Safe alternative

```
def safe_append(value: int, lst: List[int] = None) -> None:  
    if lst is None:  
        lst = []  
    lst.append(value)  
    return lst  
  
safe_append(3, [1, 2])  
[1, 2, 3]
```

```
safe_append(4)
```

```
[4]
```

```
safe_append(5)
```

```
[5]
```

If you want a new list each time, then allocate a new list each time the default argument (None) is passed.

Source: <http://docs.python-guide.org/en/latest/writing/gotchas/>



# Variable number of parameters

We sometimes want to accept a variable number of parameters

```
def max2(v1: int, v2: int) -> int:  
    return max(v1, v2)  
def max3(v1: int, v2: int, v3: int) -> int:  
    return max(v1, v2, v3)  
def max4(v1: int, v2: int, v3:int, v4:int) -> int:  
    return max(v1, v2, v3, v4)
```

```
print("hello", "world")
```

```
def max_n(*args):  
    return max(args)
```

```
hello world
```

```
max_n(2, 1, 5, 4, 6)
```

How many versions  
do we need?

Print accepts a variable number  
of arguments. Can we do that?

Accept an arbitrary number of  
arguments and assign them to a  
tuple with name 'args'



# Keyword arguments

Python's `**kwargs` allows passing arbitrary keywords and values to a function

Specify key and value in function call

```
def keywords(**kwargs: Any):  
    for key, value in kwargs.items():  
        print(f"key: {key} value: {value}")
```

```
keywords(name='Jim', age='ancient', happy=True)
```

```
key: name value: Jim  
key: age value: ancient  
key: happy value: True
```

The order of the parameters is not defined but you can access by key

```
keywords(CWID=12345, Course='SSW 810')
```

```
key: CWID value: 12345  
key: Course value: SSW 810
```



# Passing any type of arguments

To capture all possible arguments to a function

```
def parameters(*args: Any, **kwargs: Any):  
    print(f"args: {args}")  
    print(f"kwargs: {kwargs}")  
  
parameters(1, 2, 3, name='Fred', age=20)
```

args: (1, 2, 3)

args is stored as a tuple

kwargs: {'name': 'Fred', 'age': 20}

kwargs is stored as a dict

And now  
for something  
completely different...





# Python modules

Python source code files may be used by other programs

Python includes many standard packages

You can make your own packages

PyPI (Python Package Index) offers 200,000+ packages

<https://pypi.python.org/pypi>

Your program may be split across multiple files

Use import to access shared resources





# Installing Python Packages

Python comes with many packages preinstalled

You may need to install a few packages from time to time

See <https://packaging.python.org/tutorials/installing-packages/>

```
$ pip install package_name
```

```
$ pip install prettytable
```

# 3 Ways to import Python modules

import ***module\_name*** [as *alias*]

from ***module\_name*** import *object<sub>1..n</sub>*

from ***module\_name*** import \*

Don't do this!!!

This option blindly loads everything from the module into your namespace which may cause conflicts or worse





# Python modules

`import module_name [as alias]`

All of the objects from **module\_name** are available to your program, e.g. functions, global variables, etc.

Use dot notation to access objects within a module

```
import random
import math as m
from random import randint

r: float = random.random()
sqrt: float = m.sqrt(9)
i: int = randint(0, 100)
```

Make all objects in the “random” module available to your program

Call the random() function defined within the “random” module

Call the sqrt() function defined within the “math” module

# Python modules

```
from module_name import *
```

Don't EVER use this form!!!

Imports everything from module\_name into your module BUT you can't control what is or is not included

Dot notation is not needed to access objects within a module

Beware that objects can be overwritten

```
# evil_module.py
def print(*args: Any) -> None:
    raise ValueError("You've been hacked!!!")
```



```
from evil_module import *
print("hello world")
```

```
ValueError: You've been hacked!!!
```



# Python modules

`from module_name import objects`

Makes specified objects from ***module\_name*** available to your program, e.g. functions, global variables, etc.

Dot notation is **not** needed to access objects within a module

Beware that objects can be overwritten

```
from random import randint  
i: int = randint(0, 100)
```

`random.randint()` is available ***without*** dot notation. No other objects from random have been imported, only ***randint()***

# Namespaces: Which X is which?

Each module is associated with a single file

Each module has a namespace and all of the objects in that module are in that namespace



# Namespaces: Which x is which?

Module: mod1.py

```
n: int = 42
def foo() -> None:
    print(f"mod1.foo: n={n}")
```

```
import mod1, mod2
n: int = 3
def foo() -> None:
    print(f"main.foo: n={n}")
```

```
foo()
```

```
main.foo: n=3
```

```
mod1.foo()
```

```
mod1.foo: n=42
```

```
mod2.foo()
```

```
mod2.foo: n=17
```

Module: mod2.py

```
n: int = 17
def foo() -> None:
    print(f"mod2.foo: n={n}")
```





# What could go possibly go wrong?

```
from random import random
print(f"before: {random()}")

def random() -> str:
    return f"Oops! Random() has been defined"

print(f"after: {random()}")
```

before: 0.1415730749943951  
after: Oops! Random() has been defined

**OOPS**

# Breaking projects into files

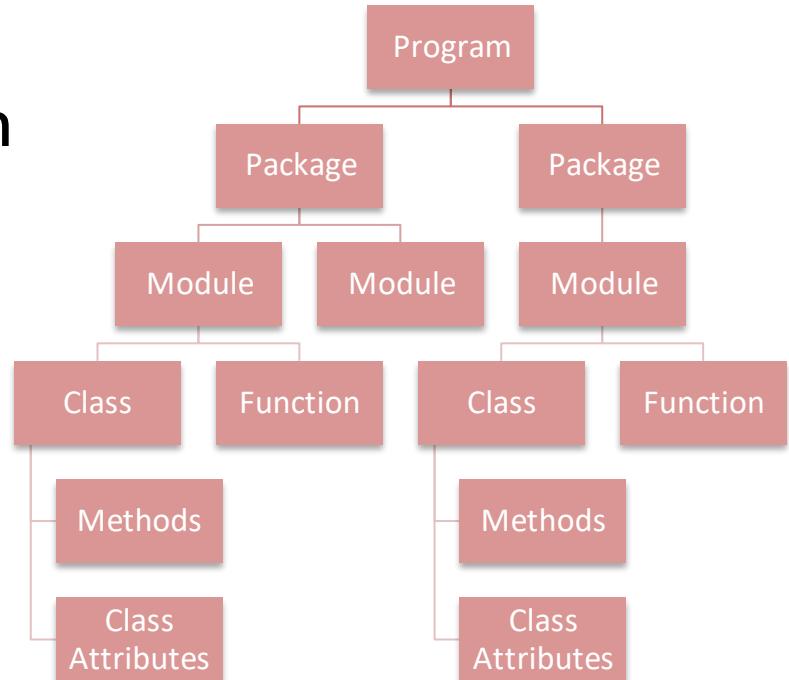
Larger programs typically include multiple files

Projects are frequently written by teams

Different developers write different components

Integrate the files together into a solution

You can make your own packages



# Projects with multiple files

## modules file0.py

```
def file0_api(s: str) -> None:
    print(f"file0_api: {s}")
    file0_local(s)

def file0_local(s: str) -> None:
    print(f"file0_local: {s}")
```

## modules file1.py

```
def file1_api(s: str) -> None:
    print(f"file1_api: {s}")
    file1_local(s)

def file1_local(s: str) -> None:
    print(f"file1_local: {s}")
```

```
from file0 import file0_api
from file1 import file1_api
file0_api("main calling file0_api")
```

```
file0_api: main calling file0_api
file0_local: main calling file0_api
```

```
file1_api("main calling file1_api")
```

```
file1_api: main calling file1_api
file1_local: main calling file1_api
```

```
file1_local("main calling file1_local")
```

```
NameError: name 'file1_local' is not defined
```

Imported objects are accessible

Only imported objects are accessible



# Useful Python modules

We'll explore a few useful packages

Python includes many standard packages

sys, random, math, time, datetime, os, ...

You can create and contribute your own packages





# Sys module

## Operating system functionality

```
import sys
```

Object	Semantics
sys.exit()	Terminate the process gracefully
sys.modules	See the modules that have been loaded
sys.path	Search path for modules
sys.stdin, sys.stdout, sys.stderr	Control input, output, and error files
sys.argv	Command line arguments
sys getopt	Parse command line arguments
Many others	

<https://docs.python.org/3/library/sys.html>



# Command line arguments: sys.argv

```
$ cat 08-argv.py
```

```
import sys
```

```
print(f"Found {len(sys.argv)} arguments")
```

```
print(f"argv = {sys.argv}")
```

```
$ python 08-argv.py hello world
```

```
Found 3 arguments
```

```
argv = ['08-argv.py', 'hello', 'world']
```

**getopt, argparse, ... modules help to parse command line arguments**



# Random number module

## Pseudo-random numbers

```
import random
```

Object	Semantics
random.seed(a=None)	Initialize the random number generator. You can use this to reproduce the same random number sequence.
random.randint(a,b)	Return a random integer, n, where $a \leq n \leq b$
random.random()	Return a random float f, where $0 \leq f \leq 1.0$
random.choice(sequence)	Return a random element from sequence
Many others	

<https://docs.python.org/3/library/random.html>



# Math module

## Math functions

```
import math
```

Object	Semantics
math.ceil(n)	Return ceiling(n)
math.factorial(n)	$n!$
math.floor(n)	Largest integer $\leq n$
math.trunc(n)	Truncate real values to integers
math.exp(n)	Return $e^{**n}$
math.pow(x,y)	Return $x^{**y}$
math.sin(x), math.cos(x), math.tan(x), ...	

<https://docs.python.org/3/library/math.html>

# Time module

Use time module to measure time spent running Python code

Helpful for performance analysis of algorithms

Task: compare the performance of two implementations of factorial



- `math.factorial(n)`
- recursive implementation of factorial

# Performance comparison

```

from time import process_time
import math
TARGET: int = 50
REPS: int = 50000
def factorial(n: int) -> int:
    """ calculate n! recursively """
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

start_recursive: float = process_time() # start time
for n in range(REPS):
    _ = factorial(TARGET)
stop_recursive = process_time() # end time

start_math: float = process_time() # start time
for n in range(REPS):
    _ = math.factorial(TARGET)
stop_math = process_time() # end time

print(f"Recursive solution: {stop_recursive - start_recursive:.4f} secs")
print(f"Math solution: {stop_math - start_math:.4f} secs")

```

Recursive solution: 0.6692 secs

Math solution: 0.0593 secs

## Timing Results

Recursive: 0.6692 secs

math.factorial: 0.0593 secs

Note time at start of test

Note time at end of test

# datetime module

Date arithmetic is a common task

Harder than you might think:

3 days from now

4 months ago

Last 90 days

`date2 – date1`

...

<https://docs.python.org/3/library/datetime.html>





# Reading and writing dates

Dates take many forms

8/29/16      Aug 29, 2016      29/8/2016

`datetime.datetime.strptime()` parses strings as dates

`datetime.datetime.strftime()` formats dates

<https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior> has a complete list of many alternatives



# Reading and writing dates

```
from datetime import datetime

date1: str = "1 JAN 2020"
date2: str = "July 31, 2020"
date3: str = "8/29/19"

dt1: datetime = datetime.strptime(date1, "%d %b %Y")
dt2: datetime = datetime.strptime(date2, "%B %d, %Y")
dt3: datetime = datetime.strptime(date3, "%m/%d/%y")

print(f"dt1: {dt1}")
print(f"dt2: {dt2.strftime('%a, %b %d, %Y')}")"
print(f"dt3: {dt3.strftime('%m/%d/%y')}")
```

```
dt1: 2020-01-01 00:00:00
dt2: Fri, Jul 31, 2020
dt3: 08/29/19
```

# Date arithmetic

```
from datetime import datetime, timedelta

date1: str = "1 JAN 2020"
date2: str = "July 31, 2020"

dt1: datetime = datetime.strptime(date1, "%d %b %Y")
dt2: datetime = datetime.strptime(date2, "%B %d, %Y")

order: str = 'before' if dt1 < dt2 else 'after'
delta: datetime = dt2 - dt1
print(f"{dt1:%m/%d/%y} occurs {delta.days} days {order} {dt2:%m/%d/%y}")

num_days: int = 3
dt3: datetime = dt1 + timedelta(days=num_days)
print(f"{num_days} days after {dt1:%m/%d/%y} is {dt3:%m/%d/%y}")

dt4: datetime = dt1 - timedelta(days=num_days)
print(f"{num_days} days before {dt1:%m/%d/%y} is {dt4:%m/%d/%y}")
```

01/01/20 occurs 212 days before 07/31/20

3 days after 01/01/20 is 01/04/20

3 days before 01/01/20 is 12/29/19



# os module

## Access to the underlying operating system

```
import os
```

Object	Semantics
os.chdir(path)	Set current working directory to path
os.getcwd()	Get current working directory
os.listdir(path=':')	Return a list of files in the specified directory
os.path.join(dir, file_name)	Create a path from a directory and file name
os.path.basename(path)	Return the file name at the base of a path

These are helpful when reading/writing files

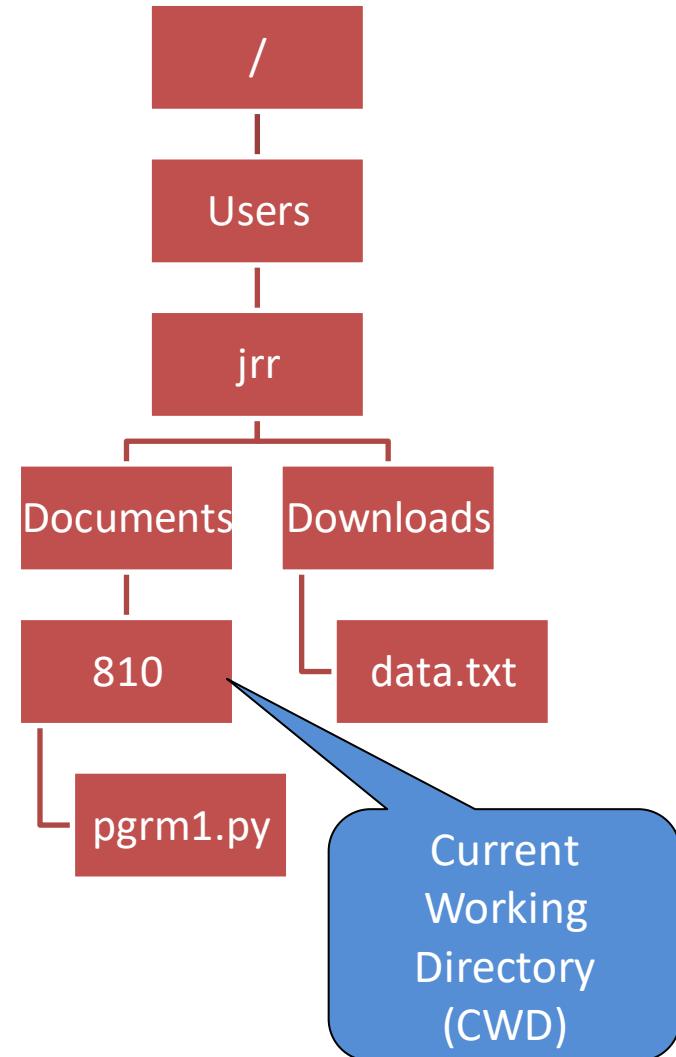
<https://docs.python.org/3/library/os.html>

# Working directories

Python runs in the Current Working Directory (CWD)

Typically the directory holding the .py file being run

Must specify full path or change CWD to access a file in a different directory





# os.path.join(dir, file)

```
# DO THIS
import os

dir: str = "/Users/jrr/Downloads"
file: str = "data.txt"
path: str = os.path.join(dir, file)
path

'/Users/jrr/Downloads/data.txt'
```

os.path.join(dir, file)  
does the right thing  
across operating  
systems and if path  
ends with or without  
trailing '/'

```
# DON'T do this
path2: str = dir + file
path2

'/Users/jrr/Downloadsdata.txt'
```



# PrettyTable module

PrettyTable provides a convenient to format tables

Say we want to print the data in a table

CWID	FirstName	LastName
10001001	Nanda	Koka
10001002	Sujit	Behara
10001003	Fei	Hou



# PrettyTable example

```
from prettytable import PrettyTable
people: List[Tuple[str, str, str]] = [
    ('1234', 'Nanda', 'Koka'),
    ('1245', 'Fei', 'Hou'),
    ('4321', 'Maha', 'Aldrisi'),
]
pt: PrettyTable = PrettyTable(field_names=['CWID', 'First Name', 'Last Name'])
for cwid, fname, lname in people:
    pt.add_row([cwid, fname, lname])

print(pt)
```

CWID	First Name	Last Name
1234	Nanda	Koka
1245	Fei	Hou
4321	Maha	Aldrisi



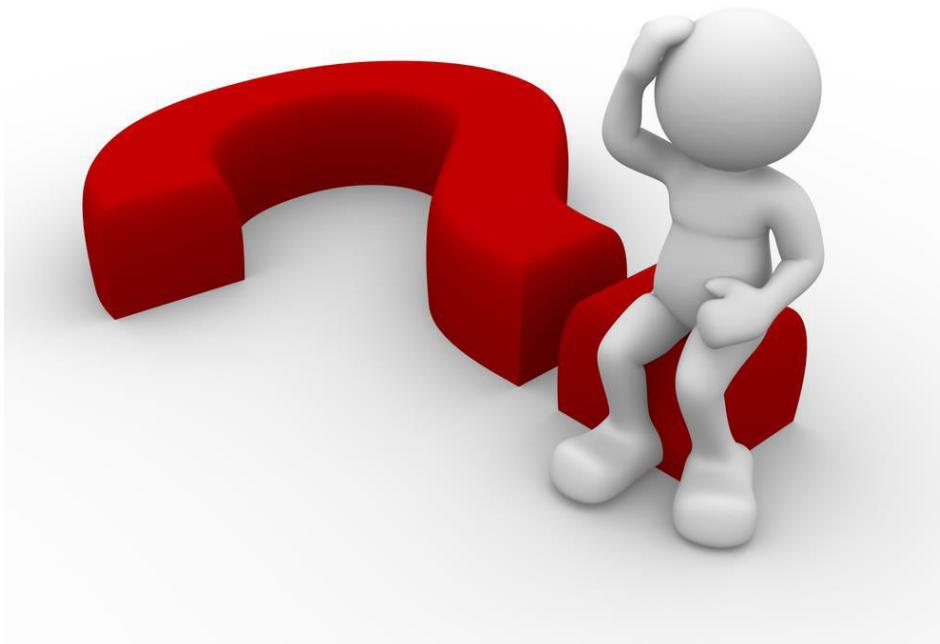
# Modules summary

There are many useful Python modules

Check PyPi for existing modules before writing  
your own

<https://pypi.python.org/pypi>

# Questions?

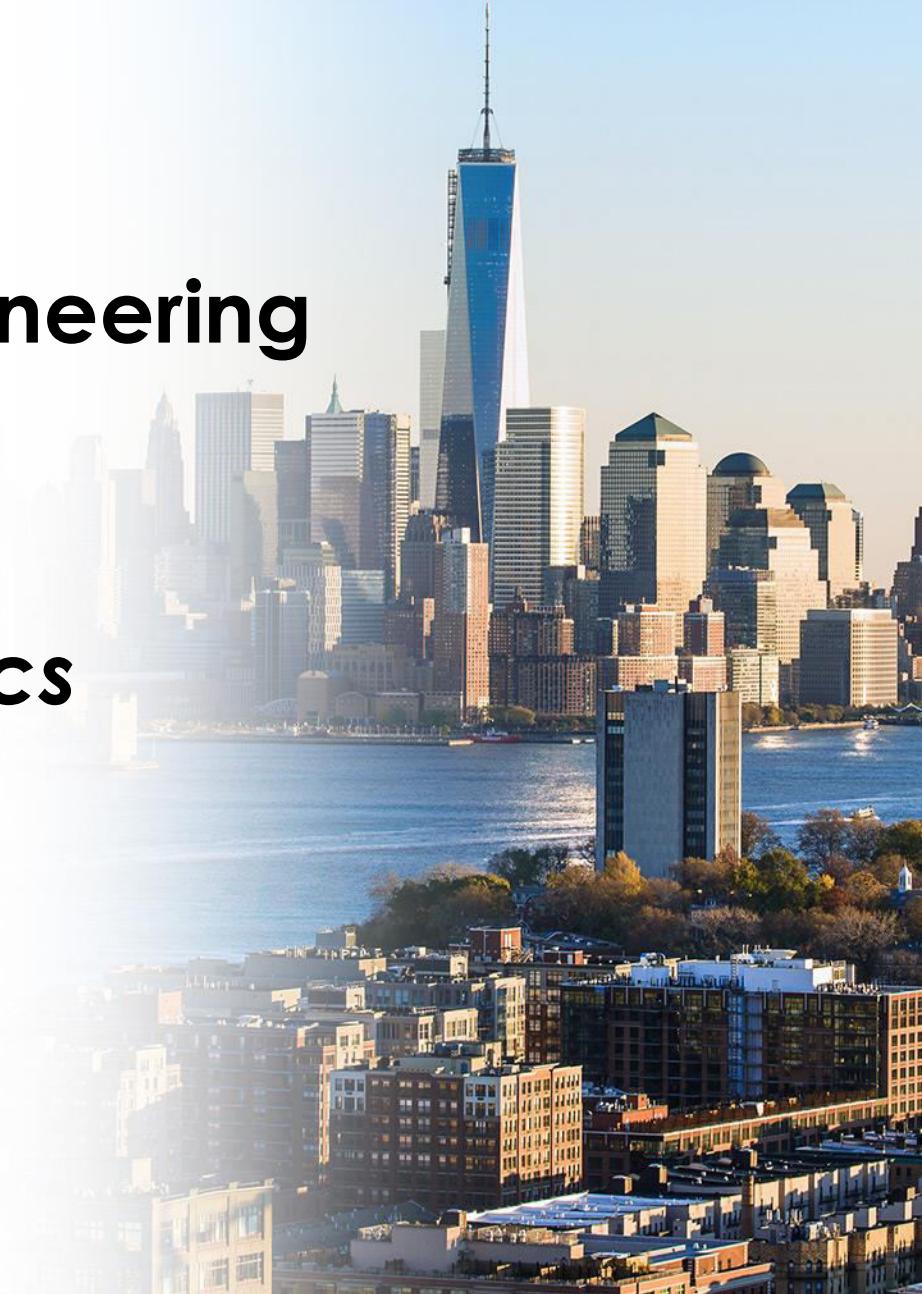




# SSW-810: Software Engineering Tools and Techniques

## *Advanced Python Topics*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises



# Today's topics

Decorators

Encapsulation

@property, @attribute.setter

Polymorphism

Dynamic typing with Duck Typing

Class attributes

Static methods



@decorator  
python™





# Decorators

Decorators allow us to **extend** the behavior of a function **without modifying** the function

**Wrapping** extra functionality before and after a function

Examples:

Time a function call

Track how many times a function is called

Memoize a function to use previous results

Mock a function call while testing

Web frameworks, e.g. Flask

<https://wiki.python.org/moin/PythonDecoratorLibrary>



# Python functions are first-class objects

## ADVANCED/OPTIONAL

Functions can be assigned to variables and passed as arguments to other functions

```
from typing import Callable, Any
TFunc = Callable[..., Any] # a function taking any arguments and returning Any value

def hello() -> str:
    return "Hello world!"

alias: TFunc = hello
alias()

'Hello world!'
```

*alias is another name for hello*



# Python functions are first-class objects

## ADVANCED/OPTIONAL

Functions can be assigned to variables and passed as arguments to other functions

```
TFunc = Callable[..., Any]
```

```
def hello() -> str:  
    return "Hello world!"
```

Call the function passed to  
call\_func

```
def call_func(func: TFunc) -> Any:  
    return func()
```

```
call_func(hello)
```

```
'Hello world!'
```

Pass function *hello* to  
call\_func as an argument



# Nested functions

## ADVANCED/OPTIONAL

Functions can be nested inside other functions

```
def outer() -> None:  
    def inner() -> None:  
        print("inner()")  
  
    print("outer() before inner()")  
    inner()  
    print("outer() after inner()")
```

```
outer()
```

```
outer() before inner()  
inner()  
outer() after inner()
```

inner() is visible within  
outer()



# Nested functions

**ADVANCED/OPTIONAL**

Nested functions are visible only inside the nesting function

```
def outer():
    def inner():
        print("inner()")

    print("outer() before inner()")
    inner()
    print("outer() after inner()")

inner()
```

inner() is visible **only**  
within outer()

NameError: name 'inner' is not defined

# Decorators: Wrapping functions

## ADVANCED/OPTIONAL

```
TFunc = Callable[..., Any]
```

```
def hello() -> str:
    print("Hello world!")
    return "Hi!"
```

arbitrary function

```
def my_decorator(func: TFunc) -> TFunc:
    def wrapper() -> Any:
        print("wrapper before calling func")
        result = func() # call func and store return value
        print("wrapper after calling func")
        return result # return value returned by func

    return wrapper # return the inner wrapper function
```

```
hello = my_decorator(hello)
hello()
```

```
wrapper before calling func
Hello world!
wrapper after calling func
```

Decorator returns the *inner* wrapper function

hello is now the decorated hello so calling hello() calls my\_decorator() and that calls the original hello()

'Hi!'

Value returned from decorated hello()



# @decorator syntax

## ADVANCED/OPTIONAL

```
TFunc = Callable[..., Any]

def my_decorator(func: TFunc) -> TFunc:
    def wrapper() -> Any:
        print("wrapper before calling func")
        result = func() # call func and store return value
        print("wrapper after calling func")
        return result # return value returned by func

    return wrapper # return the inner wrapper function
```

```
@my_decorator
def hi() -> str:
    print("Saying 'hi!'")
    return "Hi there!"
```

```
hi()
```

```
wrapper before calling func
Saying 'hi!'
wrapper after calling func
'Hi there!'
```

Syntactic shortcut for  
hi = my\_decorator(hi)

Now we can call the  
*decorated* hi() which calls the  
*original* hi() function



# One decorator, many functions

## ADVANCED/OPTIONAL

We can apply the same decorator to many functions

```
TFunc = Callable[..., Any]

def my_decorator(func: TFunc) -> TFunc:
    def wrapper() -> Any:
        print("wrapper before calling func")
        result = func() # call func and store return value
        print("wrapper after calling func")
        return result # return value returned by func

    return wrapper # return the inner wrapper function

@my_decorator
def hola() -> str:
    return "Hola!"

@my_decorator
def hey_there() -> str:
    return "Hey!"
```

Decorate many functions  
with the same decorator



# Decorating functions with arguments

## ADVANCED/OPTIONAL

```
TFunc = Callable[..., Any]
```

```
def decorate_args(func: TFunc) -> TFunc:  
    def wrapper(*args: Any, **kwargs: Any) -> TFunc:  
        print('decorate_args: before func')  
        result: Any = func(*args, **kwargs)  
        print('decorate_args: after func')  
        return result  
  
    return wrapper
```

wrapper() accepts any number of args and keyword args

Call func with args and kwargs

Return the result of calling func from wrapper

```
@decorate_args  
def plus2(n: int) -> int:  
    print(f"inside plus2({n})")  
    return n + 2
```

```
plus2(1)
```

```
decorate_args: before func  
inside plus2(1)  
decorate_args: after func
```



# pp\_fcall() – see function arguments

## ADVANCED/OPTIONAL

```
from typing import Any
def pp_fcall(func: TFunc, *args: Any, **kwargs: Any) -> str:
    """ return a string with the call details """
    args: List[Any] = [repr(a) for a in args]
    args += [f"{var}={repr(val)}" for var, val in kwargs.items()]
    return f"{func.__name__}({', '.join(args)})"
```

```
def foo(*args, **kwargs) -> None:
    print('hello')
```

```
print(pp_fcall(foo, 'a', 1, 2, bar='b', baz='c', bong=42))
```

```
foo('a', 1, 2, bar='b', baz='c', bong=42)
```

print details of arbitrary function call



# Execution time decorator

## ADVANCED/OPTIONAL

Accept any number/type of arguments

```
def timeit(func: TFunc) -> TFunc:  
    def wrapper(*args: Any, **kwargs: Any) -> Any:  
        start: float = time()  
        result: Any = func(*args, **kwargs)  
        end: float = time()  
        print(f"timeit: {pp_fcall(func, *args, **kwargs)} returns {result} in {end - start:.5f} secs")  
        return result  
  
    return wrapper  
  
@timeit  
def hangout(n: int) -> int:  
    sleep(n)  
    return n  
  
hangout(1)  
hangout(2)
```

Print function signature,  
return value, and  
elapsed time

```
timeit: hangout(1) returns 1 in 1.00479 secs  
timeit: hangout(2) returns 2 in 2.00505 secs
```



# decorators: execution time

## ADVANCED/OPTIONAL

```
@timeit  
def fact_recursive(n: int) -> int:  
    if n == 1:  
        return 1  
    else:  
        return n * fact_recursive(n - 1)  
  
fact_recursive(10)
```

Reuse the timeit decorator for many functions

```
timeit: fact_recursive(1) returns 1 in 0.00000 secs  
timeit: fact_recursive(2) returns 2 in 0.00017 secs  
timeit: fact_recursive(3) returns 6 in 0.00020 secs  
timeit: fact_recursive(4) returns 24 in 0.00025 secs  
timeit: fact_recursive(5) returns 120 in 0.00029 secs  
timeit: fact_recursive(6) returns 720 in 0.00033 secs  
timeit: fact_recursive(7) returns 5040 in 0.00034 secs  
timeit: fact_recursive(8) returns 40320 in 0.00037 secs  
timeit: fact_recursive(9) returns 362880 in 0.00039 secs  
timeit: fact_recursive(10) returns 3628800 in 0.00042 secs  
  
3628800
```



# decorators: execution time

## ADVANCED/OPTIONAL

```
@timeit
def fact_iterative(n: int) -> int:
    result: int = 1
    for i in range(2, n + 1):
        result *= i
    return result

fact_iterative(10)
```

@timeit helps with performance analysis

```
timeit: fact_iterative(10) returns 3628800 in 0.00000 secs
```

```
3628800
```

```
fact_iterative(20)
```

```
timeit: fact_iterative(20) returns 2432902008176640000 in 0.00001 secs
```

```
2432902008176640000
```



# decorators: logging function calls

## ADVANCED/OPTIONAL

```
from datetime import datetime

def log_calls(func: TFunc) -> TFunc:
    def wrapper(*args: Any, **kwargs: Any) -> TFunc:
        print(f"log_calls: calling {pp_fcall(func, *args, **kwargs)} at",
              datetime.strftime(datetime.now(), "%m/%d/%Y %H:%M:%S"))
        return func(*args, **kwargs)
    return wrapper

@log_calls
def foo(n: int) -> int:
    return n

@log_calls
def bar(n: int) -> int:
    return n

foo(1)
bar(2)
foo(3)
foo(4)
```

@log\_calls helps with tracking function calls

```
log_calls: calling foo(1) at 03/19/2020 16:54:39
log_calls: calling bar(2) at 03/19/2020 16:54:39
log_calls: calling foo(3) at 03/19/2020 16:54:39
log_calls: calling foo(4) at 03/19/2020 16:54:39
```



# Decorators summary

Decorators allow us to **extend** the behavior of a function **without modifying** the function

**Wrapping** extra functionality before and after a function

```
@foo
```

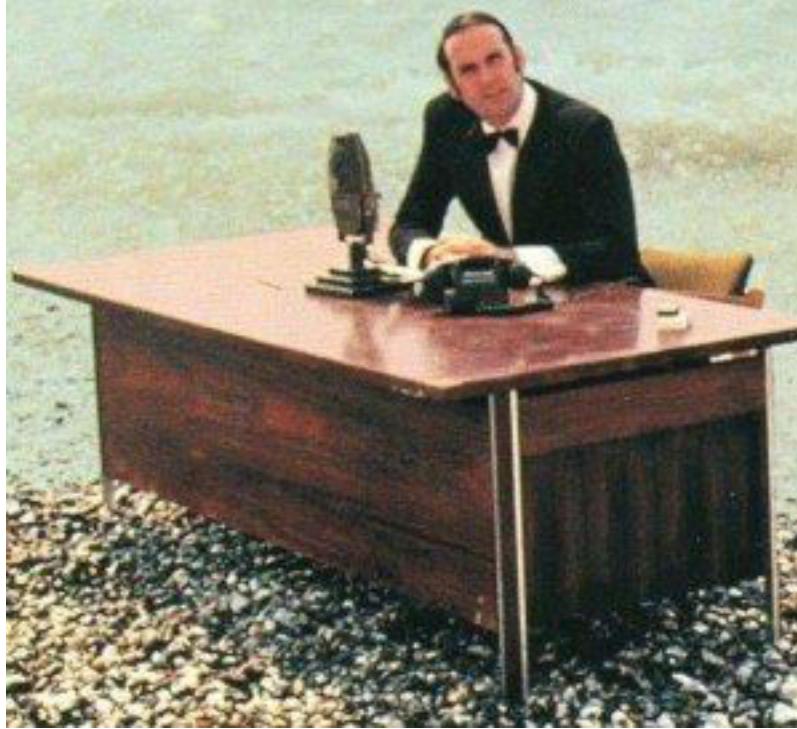
```
def bar(): ...
```

**@decorator**  
python™

<https://wiki.python.org/moin/PythonDecoratorLibrary>

I won't ask you to write a decorator  
but I do want you to understand what's  
happening when you see one

And now  
for something  
completely different...



# Encapsulation/Information Hiding: Public vs Private interfaces in classes

	Public Interfaces	Private Interfaces
Sharing	Shared freely with other modules	Not shared: details are hidden
Used by others	Safely used by other modules without concern of changes	Not available to other classes
Cost of change	New interfaces are cheap but changes may be very expensive if the change breaks other modules	Class implementation may change without impacting other modules



# **What** are we hiding **from whom** and **why**?

## **What?**

Attributes and methods critical to the proper behavior of the class  
Design decisions, e.g. critical data structures

## **From whom?**

Programmers (including the class author) with access to code  
Data privacy is **not** relevant because the code is available  
Subclasses derived from the superclass

## **Why?**

Improve encapsulation to allow the internal implementation to change without impacting the public interfaces used by other code



# Private/Protected attributes/methods

Java and C++ enforce public, protected, and private in the compiler

Python assumes that we're all responsible adults and everyone will follow the Python conventions:

name (no \_) tells the reader that name is public

\_name (single \_) tells the reader that \_name is protected

\_\_name (double \_\_) tells the reader that \_\_name is private and shouldn't be used outside the class definition





# Private/Protected in Python

name tells the reader that name is public

\_name tells reader that \_name is protected **only** by convention

\_\_name tells reader that \_\_name is private **only** by convention

```
class Employee:  
    def __init__(self, name: str, title: str, salary: int) -> None:  
        self.name: str = name  
        self._title: str = title  
        self.__salary: int = salary
```

self.attr is **public** by convention

self.\_attr is **protected** by convention

self.\_\_attr is **private** by convention

```
|  
  
fei = Employee('Fei', 'Hacker', 200000)  
print(fei.name)  
print(fei._title)  
print(fei.__salary)
```

Public and Protected attrs can be used as expected

```
Fei  
Hacker
```

Private attrs are hidden by name mangling

```
AttributeError: 'Employee' object has no attribute '__salary'
```



# Private/Protected in Python

**Don't** rely on **private** to protect sensitive information

```
class Employee:  
    def __init__(self, name: str, title: str, salary: int) -> None:  
        self.name: str = name  
        self._title: str = title  
        self.__salary: int = salary  
  
fei = Employee('Fei', 'Hacker', 200000)  
print(fei.name)  
print(fei._title)  
print(fei.__salary)
```

Fei  
Hacker

```
print(fei._Employee__salary)  
200000
```

Anyone with access to the code can  
still see all the details



# Private/Protected in Python

Use **protected** to warn readers not to rely on specific implementation details but most attributes are **public**

```
class Cart:  
    """ Store an ecommerce shopping cart of items """  
    def __init__(self) -> None:  
        """ Store the items in a protected attribute so we can change the implementation """  
        self._items: List[str] # _items is protected  
  
    def add(self, item: str) -> None:  
        """ public method to add a new item to the cart """  
        self._items.append(item)  
  
    def count(self) -> int:  
        """ public method to return number of elements in the cart """  
        return len(self._items)
```

add() and count() are public methods

The implementation may change so don't rely on protected attributes

It's safe for other code to use public attributes and methods



# e-Commerce Shopping Cart Item

## Class Item:

### Attributes

- `name`
- `price`
- `quantity`

Public so other code is free to assume that these interfaces will always be supported. The interfaces won't change.

### Methods

- `cost()`
- `__str__()`

Public so other code is free to assume that these interfaces will always be supported. The interfaces won't change.

# Java-inspired Item class methods

```
class JavaItem:  
    def __init__(self, name: str, price: float, quantity: float) -> None:  
        self._name: str = name  
        self._price: float = price  
        self._quantity: float = quantity  
  
    def get_name(self) -> str:  
        return self._name  
  
    def set_name(self, name: str) -> None:  
        self._name = name  
  
    def get_price(self) -> float:  
        return self._price  
  
    def set_price(self, price: float) -> None:  
        self._price = price  
  
    def get_quantity(self) -> float:  
        return self._quantity  
  
    def set_quantity(self, quantity: float) -> None:  
        self._quantity = quantity  
  
    def cost(self) -> float:  
        return self._price * self._quantity  
  
    def __str__(self) -> str:  
        return f"<Item: {self._quantity} {self._name} @ ${self._price:.2f} each = ${self.cost():.2f} total>"
```

```
grapes: JavaItem = JavaItem('Grapes', 2.50, 1)  
str(grapes)  
  
'<Item: 1 Grapes @ $2.50 each = $2.50 total>'
```

Getter()/Setter methods defined for all attributes

Pythonistas cringe at all the extra code to read and maintain



# Adjust price to include 20% discount

```
grapes: JavaItem = JavaItem('Grapes', 2.50, 1)
```

```
str(grapes)
```

```
'<Item: 1 Grapes @ $2.50 each = $2.50 total>'
```

```
grapes.set_price(grapes.get_price() - (grapes.get_price() * 0.20))
```

```
str(grapes)
```

```
'<Item: 1 Grapes @ $2.00 each = $2.00 total>'
```

Getters/Setters provide encapsulation at the cost of concise, readable code



# Pythonic Item class

```
class Item:  
    def __init__(self, name: str, price: float, quantity: float) -> None:  
        self.name: str = name  
        self.price: float = price  
        self.quantity: float = quantity  
  
    def cost(self) -> float:  
        return self.price * self.quantity  
  
    def __str__(self) -> str:  
        return f"<Item: {self.quantity} {self.name} @ ${self.price:.2f} each = ${self.cost():.2f} total>"
```

name, price, and quantity are public

cost() is public

```
apples = Item('Apples', 1.25, 2)  
str(apples)  
  
'<Item: 2 Apples @ $1.25 each = $2.50 total>'
```

apples.price

1.25

All attributes are available with  
object.attribute notation



# Discounting the price of Items

```
grapes: JavaItem = JavaItem('Grapes', 2.50, 1)
str(grapes)

'<Item: 1 Grapes @ $2.50 each = $2.50 total>'
```

```
grapes.set_price(grapes.get_price() - (grapes.get_price() * 0.20))
str(grapes)
```

```
'<Item: 1 Grapes @ $2.00 each = $2.00 total>'
```

```
apples: Item = Item('Apples', 1.25, 2)
str(apples)
```

```
'<Item: 2 Apples @ $1.25 each = $2.50 total>'
```

```
apples.price -= apples.price * 0.20
str(apples)
```

```
'<Item: 2 Apples @ $1.00 each = $2.00 total>'
```

Getters/Setters provide encapsulation at the cost of concise code

Which is easier to read and maintain?



# Encapsulation is a good thing...

Our **class Item** code is very popular and is widely reused in many projects

Developers have code using object.attribute notation

...BUT THEN...

This happened  
to Amazon.com

Unscrupulous customers discover that negative quantities lead to refunds

```
apples: Item = Item('Apples', 1.25, -5)  
str(apples)
```

```
'<Item: -5 Apples @ $1.25 each = $-6.25 total>'
```

Oops! We owe money to the customer for buying apples!?!?

# Encapsulation is a good thing...

We need to guard the quantity attribute against negative values

- ... maybe getters/setters aren't such a bad idea
- ... but, getters/setters bloat the code
- ... worse yet, other projects have existing code using object.attribute notation

We can't expect everyone to change all references throughout their code



What's a Pythonista to do???

Hint: we're **not** converting to Java

Hint: maybe decorators can help!



# @property, @setter

Python discourages getter/setter functions for attributes

Instead, Python encourages `obj.attribute` notation both for getting and setting attributes

What if you need extra logic around a getter or setter?

e.g. check for a valid range before returning or setting value

`@property/@setter` allows you to define a property of a class which is a function wrapper for an attribute's getter and setter with extra functionality

Execute extra code while allowing users to use `obj.attribute` notation for getting/setting values



# Changing quantity to an @property

```
class Item:
    def __init__(self, name: str, price: float, quantity: float) -> None:
        self.name: str = name
        self.price: float = price
        self.quantity: float = quantity

    def cost(self) -> float:
        return self.price * self.quantity

    def __str__(self) -> str:
        return f"<Item: {self.quantity} {self.name} @ ${self.price:.2f} each = ${self.cost():.2f} total>"
```

Let's change the semantics of  
Item.quantity to a function that  
checks for quantity < 0

```
apples: Item = Item('Apples', 1.25, 2)
str(apples)

'<Item: 2 Apples @ $1.25 each = $2.50 total>'
```



# Getting Item.quantity with @property

```
class Item:  
    def __init__(self, name: str, price: float, quantity: float) -> None:  
        self.name: str = name  
        self.price: float = price  
        self._quantity: float = quantity  
  
    @property  
    def quantity(self):  
        if self._quantity < 0:  
            return 0  
        else:  
            return self._quantity  
  
    def cost(self) -> float:  
        return self.price * self.quantity  
  
    def __str__(self) -> str:  
        return f"<Item: {self.quantity} {self.name} @ ${self.price:.2f} each = ${self.cost():.2f} total>"
```

Make \_quantity protected

@property adds a read-only getter that is invoked when Item.quantity is accessed

We can still reference self.quantity and Python calls self.quantity()

```
apples: Item = Item('Apples', 1.25, -5)  
str(apples)
```

```
'<Item: 0 Apples @ $1.25 each = $0.00 total>'
```

Quantity < 0 is 0



# What if we need a setter()?

```
apples: Item = Item('Apples', 1.25, -5)  
str(apples)  
  
'<Item: 0 Apples @ $1.25 each = $0.00 total>'
```

```
apples.quantity = 2
```

```
-----  
AttributeError  
<ipython-input-93-fd6a3df4ee35> in <module>  
----> 1 apples.quantity = 2
```

```
AttributeError: can't set attribute
```

@property decorator defines a new read-only attribute that can be accessed with dot notation

Traceback (most recent call last)

What if we need to set the value?

Oops!!!



# Adding a setter()

```
class Item:
    def __init__(self, name: str, price: float, quantity: float) -> None:
        self.name: str = name
        self.price: float = price
        self._quantity: float = quantity

    @property
    def quantity(self):
        if self._quantity < 0:
            return 0
        else:
            return self._quantity

    @quantity.setter
    def quantity(self, num: float):
        self._quantity = num

    def cost(self) -> float:
        return self.price * self.quantity

    def __str__(self) -> str:
        return f"<Item: {self.quantity} {self.name} @ ${self.price:.2f} each = ${self.cost():.2f} total>"
```

@name.setter decorator adds a method to set the value of name that can be accessed with dot notation



# Using the setter()

```
apples: Item = Item('Apples', 1.25, -5)  
str(apples)
```

```
'<Item: 0 Apples @ $1.25 each = $0.00 total>'
```

```
apples.quantity = 3  
str(apples)
```

@quantity.setter causes  
Item.quantity(self, num) to be  
called on assignment

```
'<Item: 3 Apples @ $1.25 each = $3.75 total>'
```

```
apples.price -= apples.price * 0.20  
str(apples)
```

```
'<Item: 3 Apples @ $1.00 each = $3.00 total>'
```

The legacy 20% discount  
code using dot notation  
still works without change



# @property/@attribute.setter review

We defined class Item with public attributes

Our code and code from others used dot notation to access the public attributes

We needed to change the class without breaking backward compatibility of dot notation

@property allows us to define a method to an attribute that is invoked when the value is retrieved

@attribute.setter allows to add an optional setter method



# @property/@attribute.setter summary

**@property** decorator adds a **read-only** attribute to a class that calls the associated method when the attribute is accessed with dot notation

**@attribute.setter** decorator adds a setter method to attribute where the method is invoked when the attribute is accessed with dot notation

@property and @*attribute*.setter can be used together or @property can be used alone for read-only

Can be added after code deployment to change behavior of an attribute without changing legacy code, e.g. add checks

And now  
for something  
completely different...





# Polymorphism

**Polymorphism** allows an operator to behave differently depending upon the types of the operands

```
1 + 3
```

integer arithmetic

```
4
```

```
'hello' + ' world'
```

string concatenation

```
'hello world'
```

```
[1, 2] + [3, 4]
```

list concatenation

```
[1, 2, 3, 4]
```

How does this work?

How does Python know which version of '+' to apply?

# Dynamic and Static Typing

## Statically typed

- C++, Java
- Variables are declared with a static type
- The compiler uses the static type information to support polymorphism

## Dynamically typed

- Python
- An object's type is determined at run time
- Python chooses the appropriate class method based on the type of the object



# Duck Typing

Duck typing is a dynamic typing method

Python uses duck typing to support polymorphism

If it walks like a duck and quacks like a duck, it's probably a duck

Python: if an object has a relevant property, then apply it

Duck typing supports polymorphism with or without inheritance





# Duck typing example

Recall the **Fraction** homework from earlier in the semester

We used Python's magic methods to support inline operators

```
f34 = Fraction(3, 4)
f12 = Fraction(1, 2)
(f34 + f12) == (f12 + f34)
```

True

Python determines the type of f34 and f12 at run time to choose the appropriate version of `__add__()` to invoke

We can use Duck Typing with or without inheritance to simplify our design and implementation

And now  
for something  
completely different...





# Instance Attributes vs Class Attributes

**Instance attributes** are replicated in each instance of a class

Defined in `__init__()` or added after the class is defined

```
class Employee:  
    def __init__(self, role: str) -> None:  
        self.role: str = role  
  
emp1: Employee = Employee('Developer') # each instance has its own attribute  
emp2: Employee = Employee('Tester')  
emp3: Employee = Employee('Manager')  
emp1.role
```

```
'Developer'
```

A distinct instance of role is included in each instance of class Employee

```
emp2.role
```

```
'Tester'
```



# Instance Attributes vs Class Attributes

A **Class attribute** exists exactly **once** across all instances of a class

Defined outside `__init__()` or after the class is defined

```
class Employee:
    activity: Dict[str, str] = {
        'Developer': 'coding',
        'Tester': 'testing',
        'Manager': 'golfing', }

    def __init__(self, role: str) -> None:
        self.role: str = role

    def busy(self) -> str:
        return f"I'm busy {Employee.activity[self.role]}"
```

A **single** instance of `Employee.activity` is **shared** across every instance of class `Employee`

```
emp1: Employee = Employee('Developer') # each instance has its own attribute
emp2: Employee = Employee('Tester')
emp3: Employee = Employee('Manager')
emp1.busy()
```

"I'm busy coding"

```
emp3.busy()
```

"I'm busy golfing"



# Referencing class attributes

How should we reference class attributes?

Reference **instance** attributes as instance.attribute

Reference **class** attributes as Class.attribute

Reference the class attribute as Class.attribute

```
class Counter:  
    cnt: int = 0 # class attribute  
  
    def __init__(self) -> None:  
        Counter.cnt += 1
```

```
c1: Counter = Counter()  
Counter.cnt
```

1

```
c2: Counter = Counter()  
Counter.cnt
```

2



# What could possibly go wrong?

We can also reference class attributes as `instance.class_attribute` BUT that can lead to unexpected behavior

```
class Counter:  
    cnt: int = 0 # class attribute  
  
    def __init__(self) -> None:  
        Counter.cnt += 1  
  
c1: Counter = Counter()  
Counter.cnt
```

1

Treating `c1.cnt` like an **instance** attribute, but it's a **class** attribute

`c1.cnt`

1

`Counter.cnt = 42`  
`c1.cnt`

42

Oops! `c1.cnt` changed because it is a reference to `Counter.cnt`



# What could be even more confusing?

Assigning a value to  
instance.class\_attribute creates a  
new **instance** attribute that  
takes precedence over the class  
attribute

c1.cnt = 7 creates a new instance  
attribute named cnt only in c1

```
class Counter:  
    cnt: int = 0 # class attribute  
  
    def __init__(self) -> None:  
        Counter.cnt += 1  
  
c1: Counter = Counter()  
Counter.cnt
```

1

c1.cnt

1

c1.cnt references  
class attribute in c1

```
Counter.cnt = 42  
c1.cnt
```

42

```
c1.cnt = 7  
c1.cnt
```

7

Counter.cnt

42

Now, c1.cnt  
references a new  
**instance** attribute  
in c1, not the class  
attribute



# Class Attributes and self

Self is just an instance – avoid `self.class_attribute`

```
class Confusion:  
    cls_attr: int = 3  
  
    def get(self) -> int:  
        return self.cls_attr  
  
c: Confusion = Confusion()  
print(f"should be 3: {c.get()}")  
should be 3: 3
```

Python checks first for an **instance** attribute and then for a **class** attribute

```
c.cls_attr = 4  
print(f"should be 4: {c.get()}")  
should be 4: 4
```

No **instance** attribute named 'cls\_attr' so use **class** attribute `Confusion.cls_attr`

```
Confusion.cls_attr = 5  
print(f"should be 5: {Confusion.cls_attr}")  
should be 5: 5
```

Create a new **instance** attribute named 'cls\_attr' in self

```
print(f"Which 'cls_attr'??: {c.get()}")  
Which 'cls_attr'??: 4
```

Change the value of **class** attribute `Confusion.cls_attr`

Use self's **instance** attribute 'cls\_attr'



# Static Methods

Static methods look just like other methods but are not associated with a specified class instance

No self parameter because the method is not associated with a specific instance

Instance attributes are not available – no class instance

Class attributes are available

Static methods can be replaced with procedural function calls

Static methods offer extra encapsulation and consistency

Static methods are fairly unusual

# Static Methods

```
from datetime import date

class DateCompare:
    """ Compare distance between two dates """
    conv: Dict[str, float] = {'days': 1, 'months': 30.41, 'years': 365.25}
```

`@staticmethod`

`@staticmethod` is decorator

```
def delta(dt1: date, dt2: date, units: str) -> Optional[float]:
    """ calculate and return the number of units between dates """
    if dt1 and dt2 and units in DateCompare.conv:
        return abs(dt1 - dt2).days / DateCompare.conv[units]
    return None
```

```
march_20_2020: date = date(2020, 3, 20)
jan_1_2019: date = date(2019, 1, 1)
```

No self because this is a static method

```
DateCompare.delta(march_20_2020, jan_1_2019, 'years')
```

1.215605749486653

```
DateCompare.delta(march_20_2020, jan_1_2019, 'months')
```

14.600460374876684

```
DateCompare.delta(march_20_2020, jan_1_2019, 'days')
```

444.0

And now  
for something  
completely different...





# Stevens Data Repository

HW09: Build Stevens Data Repository

Students

Instructors

Print Student Summary

Print Instructor Summary

HW10: Enhance Stevens Data Repository

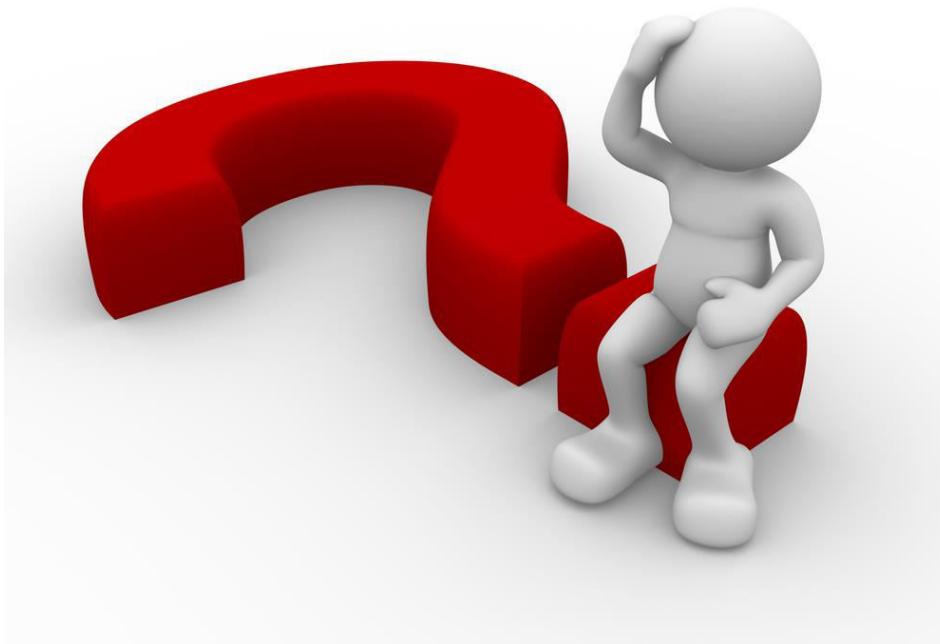
Add majors

Enhance Student Summary with remaining classes

HW11: Store data in SQL database

HW12: Present the data with a Flask web interface

# Questions?





# SSW-810: Software Engineering Tools and Techniques

*Refactoring,  
Configuration Management  
with Git and GitHub*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from

*Refactoring: Improving the Design of Existing Code* by Martin Fowler

**Pro Git** : Chacon and Straub

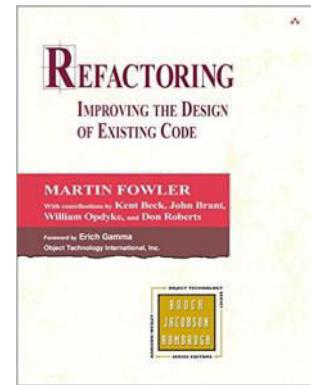
<https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf>

Getting started with GitHub

<https://guides.github.com/activities/hello-world/>

<https://guides.github.com/>

<https://www.git-tower.com/learn/>



# Overview

We don't always get software right the first time

Bad smells

Technical debt

Refactoring

Why manage file versions?

What is Configuration Management?

Tools for Configuration Management

Git

GitHub



# Software evolution

We don't always (rarely) get software right the first time

Writing the code helps to identify improvements

Respond to changes in requirements, architecture and design

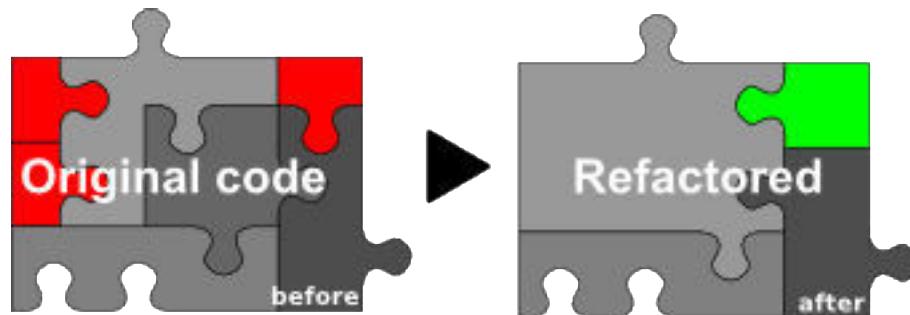
Some Agile Methods (XP) emphasize providing the simplest solution that meets all requirements, but may require later changes

How to optimize code readability and maintainability without losing functionality and reliability?



# What is refactoring?

Refactoring: changing the internal structure of software to make it **easier to understand** and **cheaper to modify** without changing its **observable behavior**



[http://www.moniro.com/my\\_pictures/product-software-refactoring.png](http://www.moniro.com/my_pictures/product-software-refactoring.png)



# What is refactoring?

Refactoring (noun): a change made to the internal structure of software to make it **easier to understand** and cheaper to modify **without changing its observable behavior**

Refactoring is **not** about performance optimization

A user should not be able to tell that the code has been refactored

A code refactoring makes the code easier to understand and maintain

Improve readability and reduce complexity

Refactoring is critical for code that changes frequently

# Why refactor?

We start with a good design and write good code to implement that design

BUT, over time the code changes to meet changes in the requirements

BUT, the design may not be updated to optimize those changes

Refactoring refreshes the design and the code to reflect an optimal solution for our latest understanding of the problem and solution

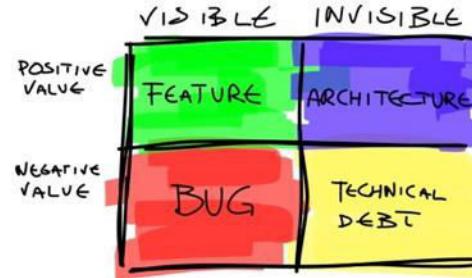


# Technical debt

Additional development, testing, and maintenance effort

Caused by

- Bad design
- Taking shortcuts, and
- Not implementing the “right” solution throughout the lifecycle



Quick hacks add technical debt

Technical debt accumulates “interest” and makes changes even harder later on

Refactoring helps to pay off "technical debt"



# Good automated test cases are critical

Refactoring replaces working code with new code

Research shows that new code is more likely to contain bugs than older code

Automated regression testing is critical

Regression testing verifies:

- The new code replicates the behavior of the old code

- The new code doesn't add any new bugs

Automated test tools, e.g. unittest, are critical

# When to refactor?

## Rule of three

1. First time, just do it
2. Second time, wince at duplication
3. Third time refactor

When you add functionality

When you need to fix a bug

As you do a code review



And now  
for something  
completely different...





# How to track changes?

You've changed your code

- Refactoring improvements

- Adding new features

What to do with earlier versions of files?

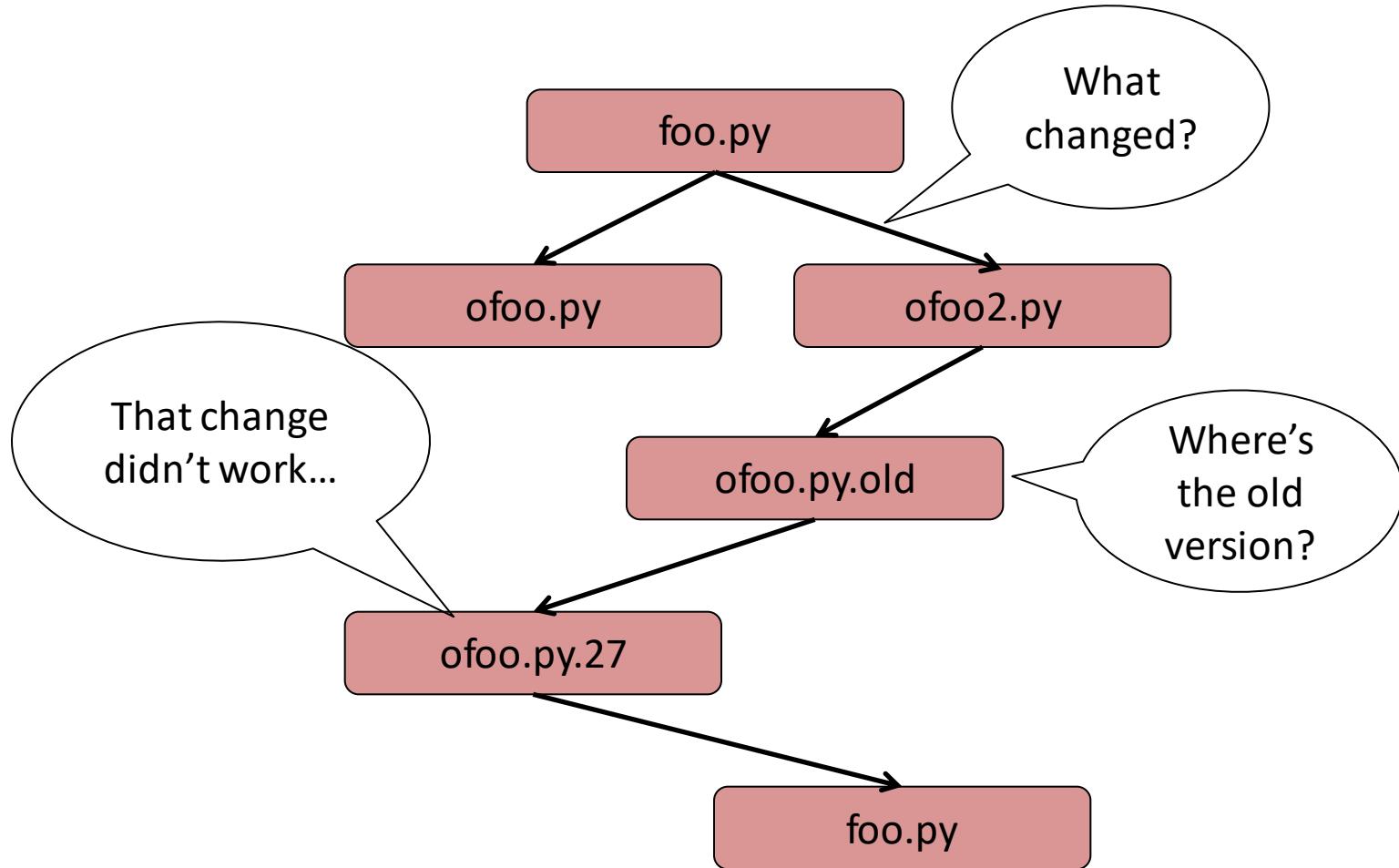
- Replace them with new versions?

- Rename file.py to ofile.py to reallyOldFile.py?

How many changes can you track in your head?

A 3D graphic of the word "CHANGE". The letters are rendered in a metallic, light gray color, except for the last letter "E" which is red. Below the word, a single red letter "C" is positioned, suggesting the continuation of the word.

# Tracking a file without tools





# What problem are we trying to solve?

You've started working on a project...

By yourself

At school with 2 other students

At a small start up company with 5 developers

At a large company with 5000 developers

How are you going to manage changes (new code and updates)?

What changed in this file?

Who made the change?

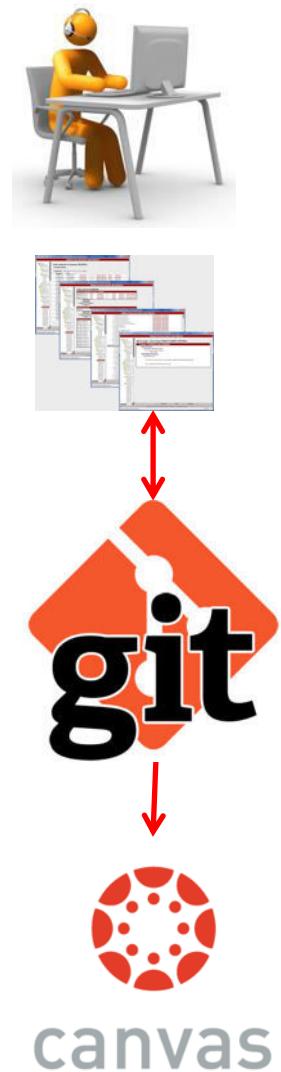
Why was the change made?

Where's the version that used to work?



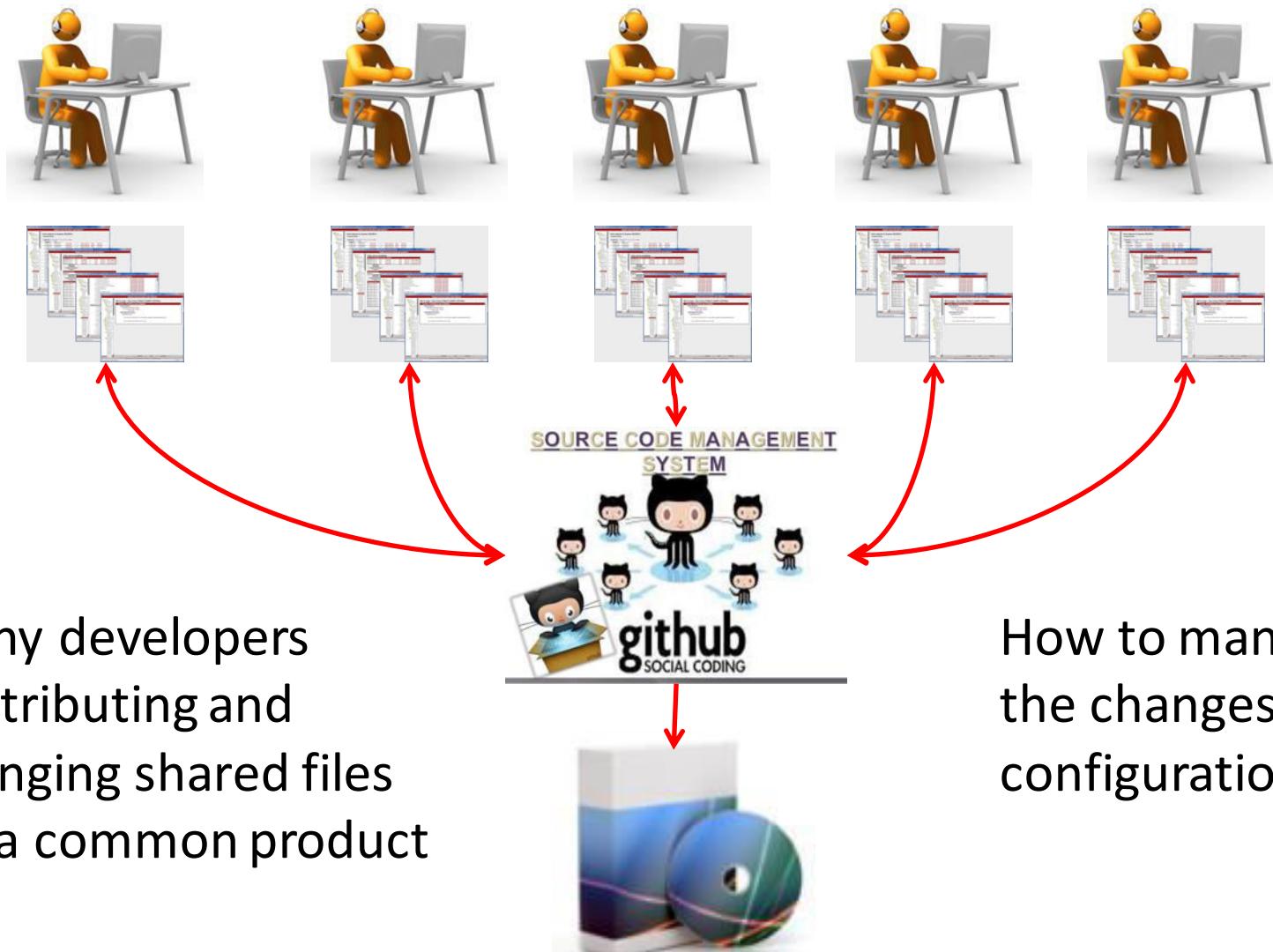
# What problem are we trying to solve?

One developer  
tracking changes  
to local files



How to manage  
the changes and  
configurations?

# What problem are we trying to solve?





# What is configuration management?

SWEBOK (Software Engineering Body of Knowledge)

[www.swebok.org](http://www.swebok.org) defines CM:

“Configuration management (CM) ... is the discipline of

- identifying the configuration of a system at distinct points in time for the purpose of systematically **controlling changes** to the configuration,
- maintaining the **integrity and traceability** of the configuration throughout the system life cycle.

The tools and processes that control and manage the artifacts and configurations of the system”

# Many solutions



★ The default for Open Source projects



# Benefits of Configuration Management

Easy access to current and previous versions of files

- Compare file versions: what changed?

- Revert to earlier versions of a file

Facilitate collaboration across team

- Support many developers working on the same files

- Create branches to explore new features

- Automatically merge multiple changes

Files can be backed up in multiple places

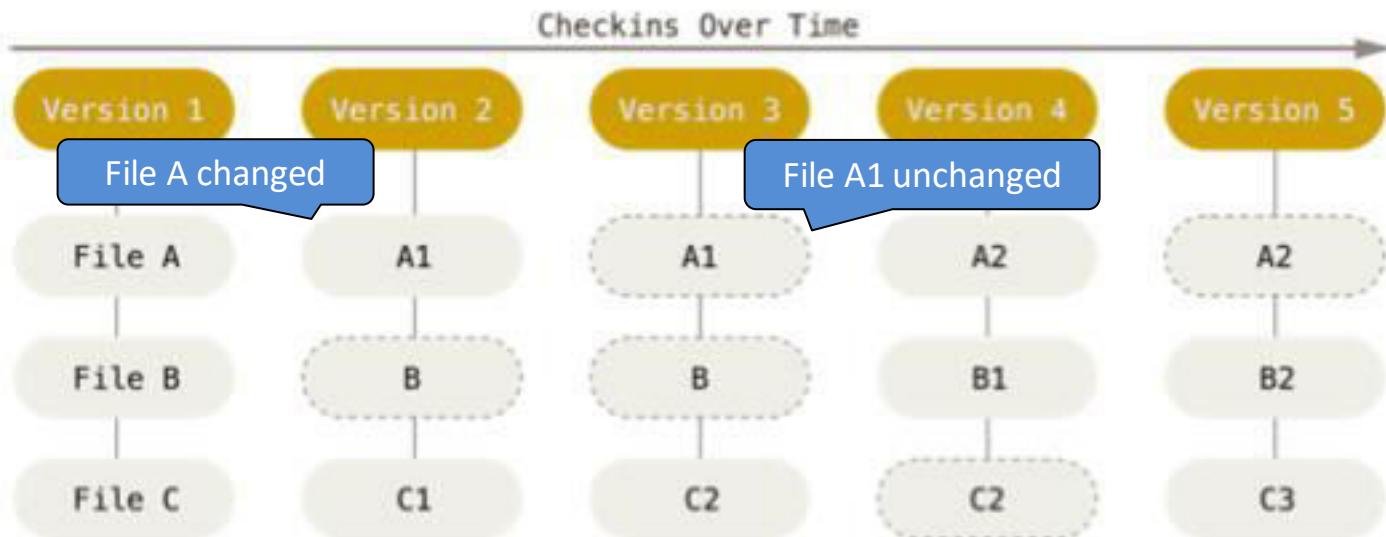
WHAT'S  
IN IT FOR  
**ME**

# Git concepts

Git maintains a stream of snapshots of the system  
Tracks changes by many contributors



Allows you to recreate the system at any earlier version



Source: <https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf>

# Git Concepts

## Repository

Files are stored in a **repository**



Developers add features or fix bugs in a **Local** repository

Local copy of the repository

Files are merged and shared in **Remote**, shared repository

Shared server, e.g. github.com

## Branch

A **branch** isolates changes in a **specific context**

Create new features, do bug fixes, experiment, ...

e.g. Master branch may represent the current product

Dev branch may represent change for a group of features

# Git Concepts

## Commits

A unit of work, i.e. a collection of related changes, written to the repository

New feature, bug fix, ...

May involve one or more files, but should represent a logical unit of functionality

Changes should be committed frequently to a branch, but only when the feature is complete



# Git Concepts

## Fetch/Push/Pull/Merge



Developers typically make changes to the local repository

We need to sync the multiple local repositories back into a branch in the common, remote repository

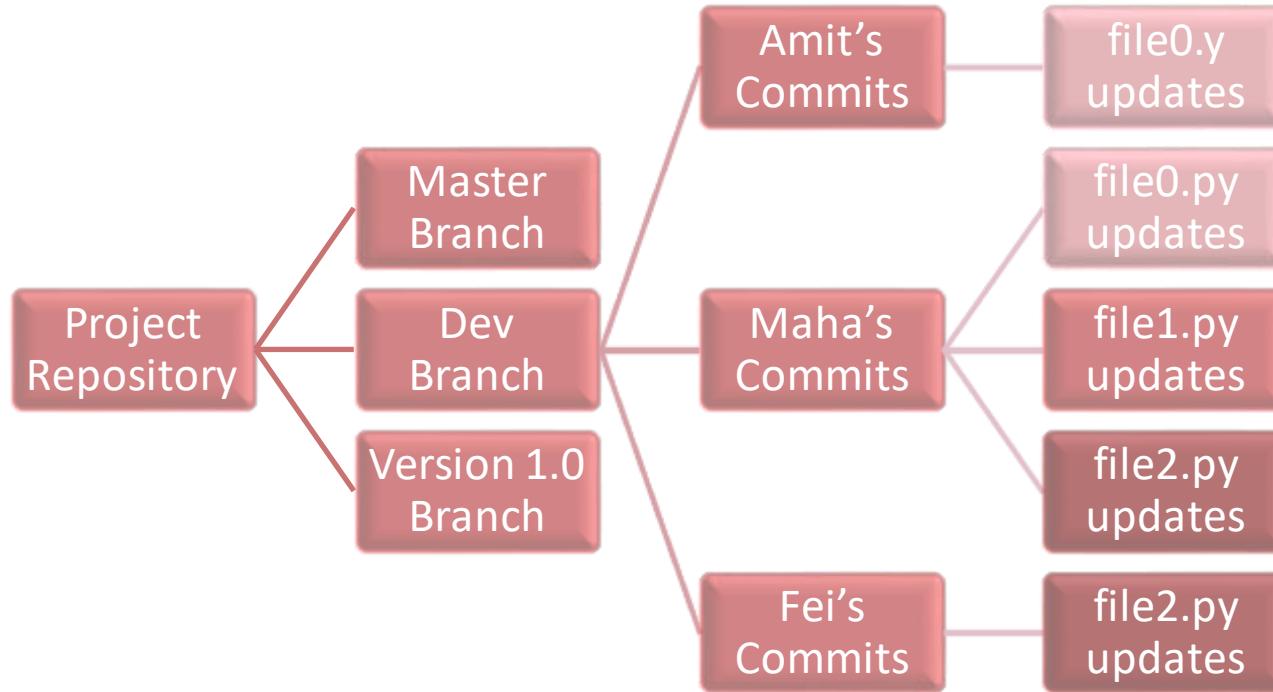
**Fetch** retrieves the latest structure from the remote repository, including all commits and latest changes

**Push** pushes the local changes up to the remote repository and merges the changes from all contributors

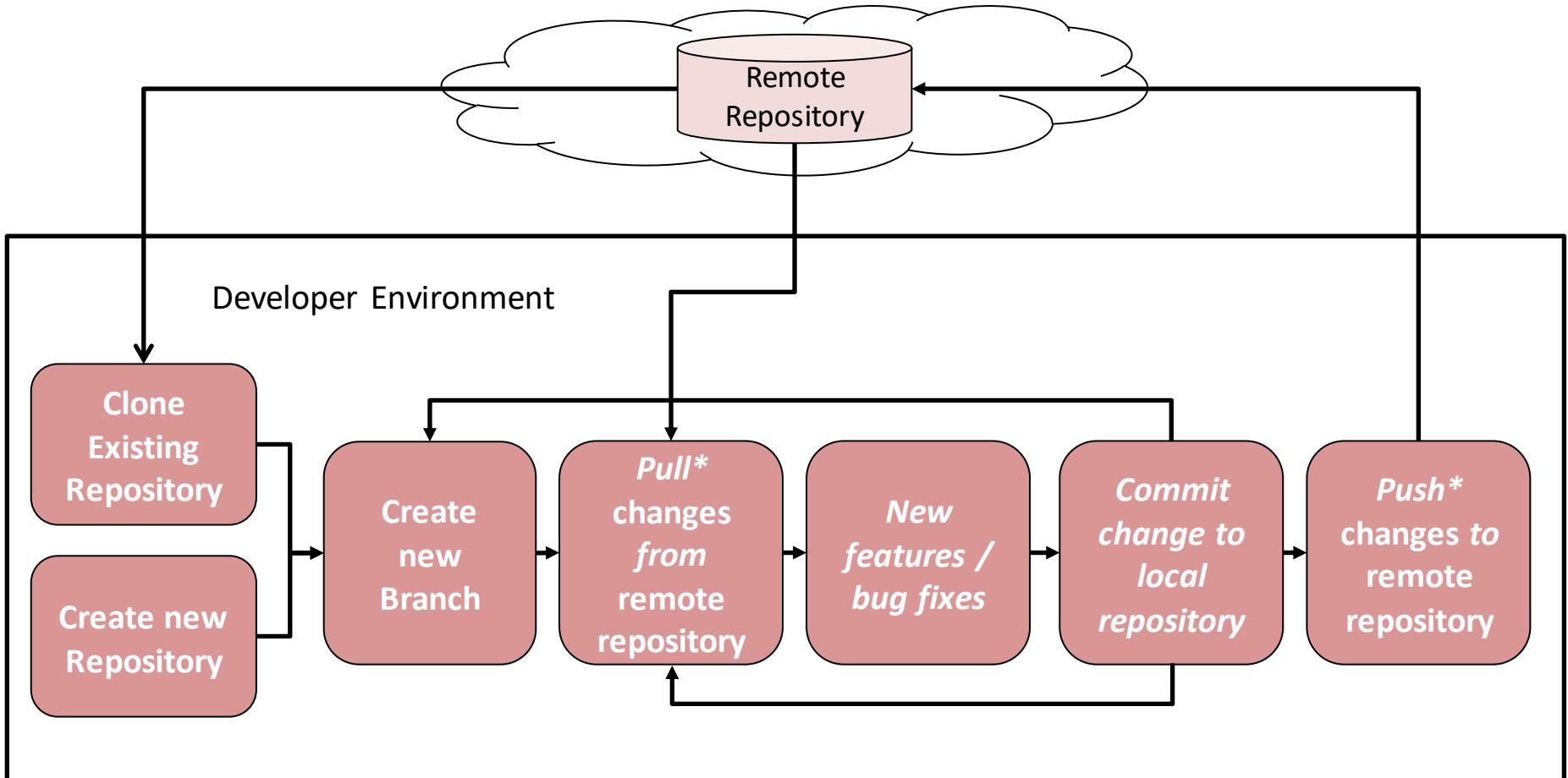
**Pull** pulls down changes from the remote repository and merges the changes into the local repository

**Merge** combines changes within or across branches

# Git Concepts



# Simplified Work Flow



\* Push and Pull merge changes across files



# 3 ways to interface with Git

```
jrr 810_Assignments $ pwd  
/Users/jrr/Downloads/810_Assignments  
jrr 810_Assignments $ git init  
Initialized empty Git repository in /Users/jrr/Downloads/810_Assignments/.git/  
jrr (master #) 810_Assignments $ ls -la  
total 0  
drwxr-xr-x  3 jrr  staff  102 Oct 29 08:02 .  
drwxr-xr-x+ 76 jrr  staff  2584 Oct 29 07:02 ..  
drwxr-xr-x  10 jrr  staff  340 Oct 29 08:02 .git  
jrr (master #) 810_Assignments $
```

## Command Line

All Git commands are available from the command line

The screenshot shows a GitHub repository named 'Prof-JR / 810\_Student\_Repository'. The repository details page is displayed, showing 1 commit, 1 branch, 0 releases, and 1 contributor. The repository name is 'SSW 810 Student Repository Project'. It contains files like .gitignore, LICENSE, README.md, and a README.rst file. A 'Clone or download' button is visible, along with options for 'Clone with SSH' and 'Use HTTPS'.

<http://github.com>

Manage your project from the GitHub website

The screenshot shows the GitHub Desktop application window. It displays a local repository named '810\_Student\_Repository' with a single branch called 'Master'. The interface includes sections for 'Changes', 'History', and 'Exchanged files'. A message at the bottom asks if the user wants to open the repository in Finder. There is also a 'Commit to master' button.

## GitHub Desktop Client, etc.

Several desktop clients are available  
Simplifies common Git commands



# Let's try it...

Walk through a common scenario...

1. Create a remote repository for a new project on github.com
2. Clone the remote repository to a local repository
3. Create a new ‘dev’ branch
4. Create a new file in the local repository
5. Commit changes locally
6. Push changes to the remote repository
7. Change the file in the remote repository
8. Pull the changes from the remote repository
9. Merge ‘dev’ and ‘master’ branch



© Can Stock Photo



# Creating a Repository: new or clone

## Create repository

Create a *new* repository for a *new* project

Unusual, except for  
new projects

Create a *local* or *remote* repository

<http://github.com> – most repositories are public to all

Shared server (private or cloud)

Remote repository provides back up

## Clone an existing repository

More common  
when joining an  
existing project

Local copy of the structure of the remote repository

Pull changes from the remote repository to local

Push local changes to the remote repository

Ideal for team projects



# Create a new GitHub Repository

A screenshot of a GitHub profile page for 'Prof-JR'. The page shows a green T-shaped profile picture, the name 'Prof-JR', and buttons for 'Add a bio' and 'Edit profile'. Below the profile picture, it says 'Overview' and lists 'Repositories 5', 'Stars 0', 'Followers 1', and 'Following 0'. A 'Popular repositories' section shows a pinned repository '810\_Student\_Repository' for 'SSW 810 Student Repository Project'. At the bottom, there's a 'Contribution settings' section showing a grid of contributions from July to June, with a legend for 'Less' (light gray), 'More' (green), and 'More' (dark green). A context menu is open in the top right corner, with 'New repository' highlighted in blue, and other options include 'Import repository', 'New gist', and 'New organization'.

Create a free  
GitHub login

Create a new  
repository



# Create a new GitHub Repository

## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner      Repository name

Prof-JR  / factorial

Great repository names are short and memorable. Need inspiration? How about [sturdy-telegram](#).

Description (optional)

Public  
Anyone can see this repository. You choose who can commit.

Private  
You choose who can see and commit to this repository.

Initialize this repository with a README  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: Python  | Add a license: None

**Create repository**

## Public Repositories

Visible to everyone  
Great for open source projects

Free of charge

Private repositories are visible only by invitation

Requires paid account

.gitignore tells Git to ignore generated files, e.g. .DS\_Store



# Create a new Repository (github.com)

The screenshot shows a GitHub repository page for 'Prof-JR / factorial'. The page has a dark header with navigation links: Pull requests, Issues, Marketplace, Explore, Watch (0), Star (0), Fork (0), and a search bar. Below the header, there's a breadcrumb trail: Prof-JR / factorial. The main content area shows a message 'No description, website, or topics provided.' with an 'Edit' button. It also shows metrics: 1 commit, 1 branch, 0 releases, and 1 contributor. A blue callout bubble points to the 'A new, empty repository' message. At the bottom, there's a list of files: 'Prof-JR Initial commit' (latest commit 49bd207 just now) and '.gitignore' (Initial commit, just now). A blue callout bubble at the bottom right points to the 'Add a README' button.

A new, empty repository

No description, website, or topics provided.

Add topics

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

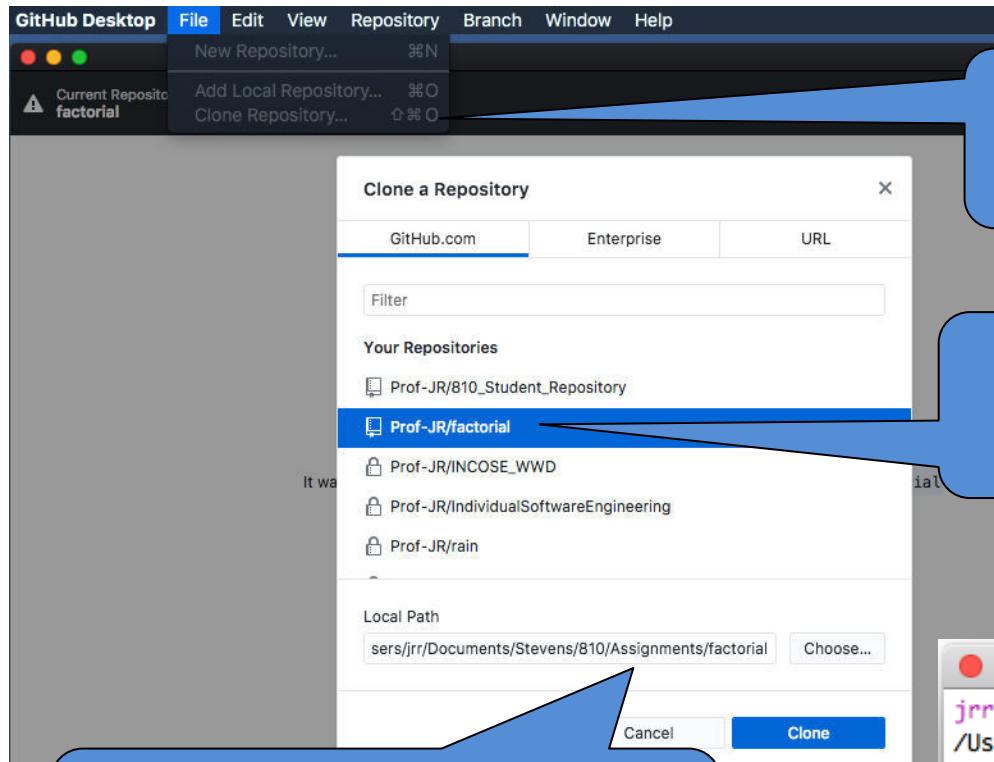
Prof-JR Initial commit Latest commit 49bd207 just now

.gitignore Initial commit just now

Add a README

You can also create new repositories in GitHub Desktop

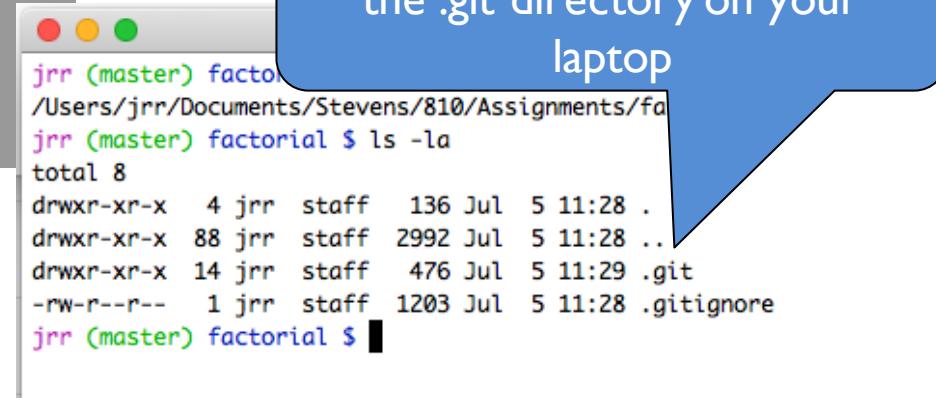
# Create local clone of the remote (GitHub Desktop)



Use **GitHub Desktop app** to clone  
the new repository to your laptop

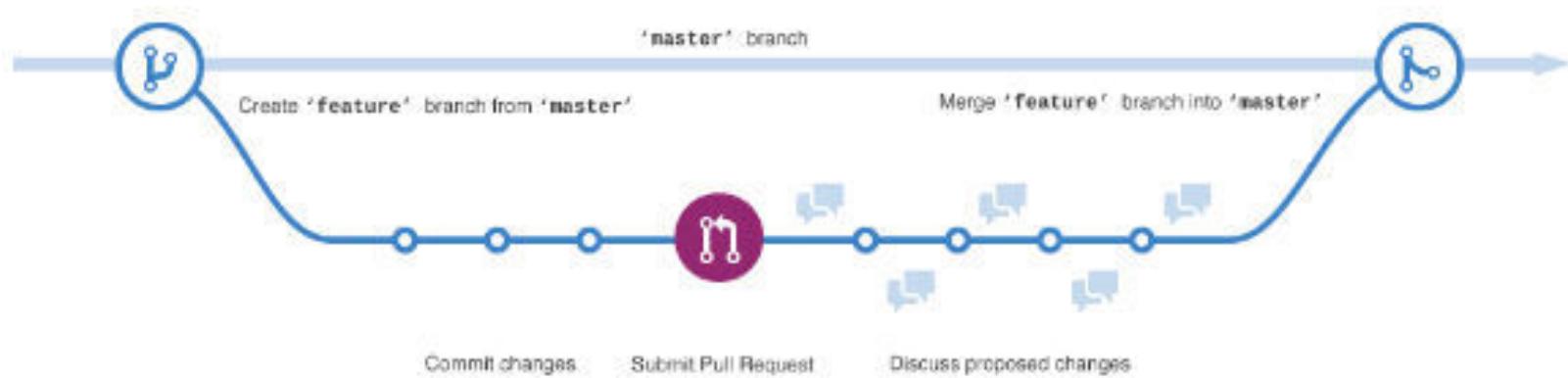
The 'factorial' repository  
hosted on github.com

Choose the **working directory**  
to store the local repository



```
jrr (master) factorial
/Users/jrr/Documents/Stevens/810/Assignments/factorial
jrr (master) factorial $ ls -la
total 8
drwxr-xr-x  4 jrr  staff   136 Jul  5 11:28 .
drwxr-xr-x  88 jrr  staff  2992 Jul  5 11:28 ..
drwxr-xr-x  14 jrr  staff   476 Jul  5 11:29 .git
-rw-r--r--  1 jrr  staff  1203 Jul  5 11:28 .gitignore
jrr (master) factorial $
```

# GitHub Branches



Branches allow you to make and test changes to the repository

Master branch is the “official” production version

The Master branch is the version deployed to customer

Feature branches allow experimenting with changes

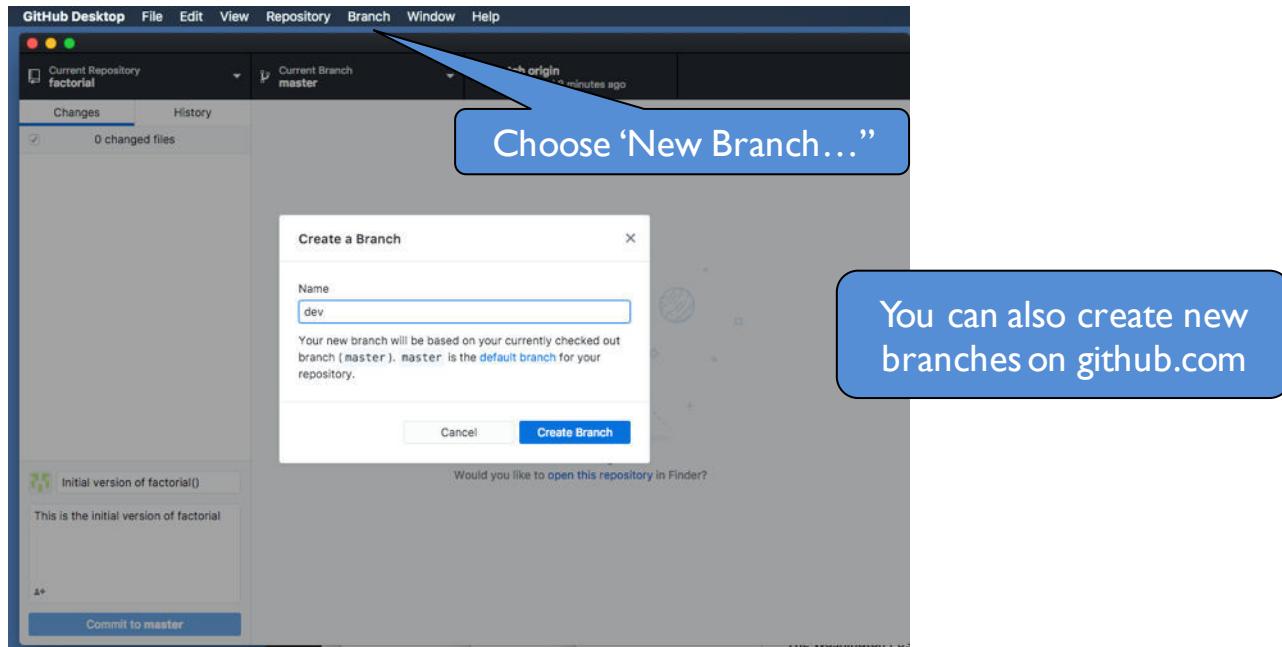
Feature branches are safe sandboxes where you develop and test code

Source: <https://guides.github.com/activities/hello-world/>

# Git Branches

Branches allow for independent development paths

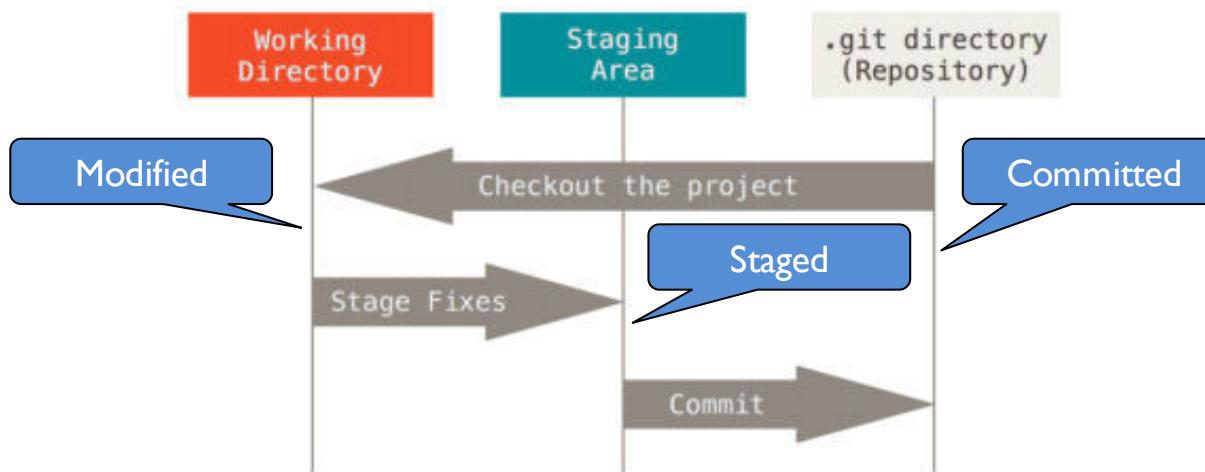
Create a new branch, e.g. **dev** where we'll add new features



# Git's three states for files

Files are in one of three states

1. **Modified** – changed locally, but **not** committed to the git repository
2. **Staged** – marked as ready to be committed to the git repository
3. **Committed** – safely stored in the repository



Source: <https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf>



# Add a file to Working Directory

The image shows two screenshots illustrating the process of adding a file to a working directory. On the left, the GitHub Desktop application interface is shown. It has a top bar with 'Current Repository' set to 'factorial' and 'Current Branch' set to 'dev'. A blue callout bubble points to the 'Publish branch' button with the text 'Ready to publish new branch to remote repository after commit'. Below this, the 'Changes' tab is selected, showing '1 changed file' named 'factorial.py'. A blue callout bubble points to this tab with the text 'VS Code, but you can use any tool to create/changes files'. The main area displays the contents of 'factorial.py' with a diff view:

```
@@ -0,0 +1,6 @@
+def factorial(n):
+    """ return n! using a recursive solution """
+    if n == 1:
+        return 1
+    else:
+        return n * 10*
```

On the right, a screenshot of VS Code is shown. It also has a top bar with 'Current Repository' set to 'factorial' and 'Current Branch' set to 'dev'. A blue callout bubble points to the status bar with the text 'Working in the 'dev' branch'. The main code editor window shows the same 'factorial.py' code as the GitHub Desktop screenshot:

```
1 def factorial(n):
2     """ return n! using a recursive solution """
3     if n == 1:
4         return 1
5     else:
6         return n * 10*
```

Below the code editor, a message in the terminal says 'Message (press Cmd+Enter to commit)'. A blue callout bubble points to this message with the text 'VS Code plugins support git'. At the bottom of the VS Code interface, there is a 'Commit to dev' button.

Added a file to the Working Directory,  
Git detects the change, GitHub  
Desktop displays the changes



# Added a file to local working directory but remote isn't aware yet...

A screenshot of a GitHub repository page for "Prof-JR / factorial". The repository has 0 stars, 0 forks, and 1 contributor. It shows 1 commit and 0 issues. A blue speech bubble points to the commit history area, containing the text: "We created the new branch locally and published the new branch to the remote repository". Another blue speech bubble points to the bottom right corner of the page, containing the text: "The remote repository isn't aware yet of the new file in the local repository".

Code Issues 0

No description, website, or Add topics

1 commit

Branch: dev New pull request

Create new file Upload files Find file Clone or download

This branch is even with master.

Pull request Compare

Prof-JR Initial commit Latest commit 49bd207 an hour ago

.gitignore Initial commit an hour ago

Help people interested in this project by adding a README.

Add a README

The remote repository isn't aware yet of  
the new file in the local repository



# git status, git add

```
jrr (dev) factorial $ git status
On branch dev
Your branch is up-to-date with 'origin/dev'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    factorial.py

nothing added to commit but untracked files present (use "git add" to track)
jrr (dev) factorial $ git add factorial.py
jrr (dev +) factorial $ git status
On branch dev
Your branch is up-to-date with 'origin/dev'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

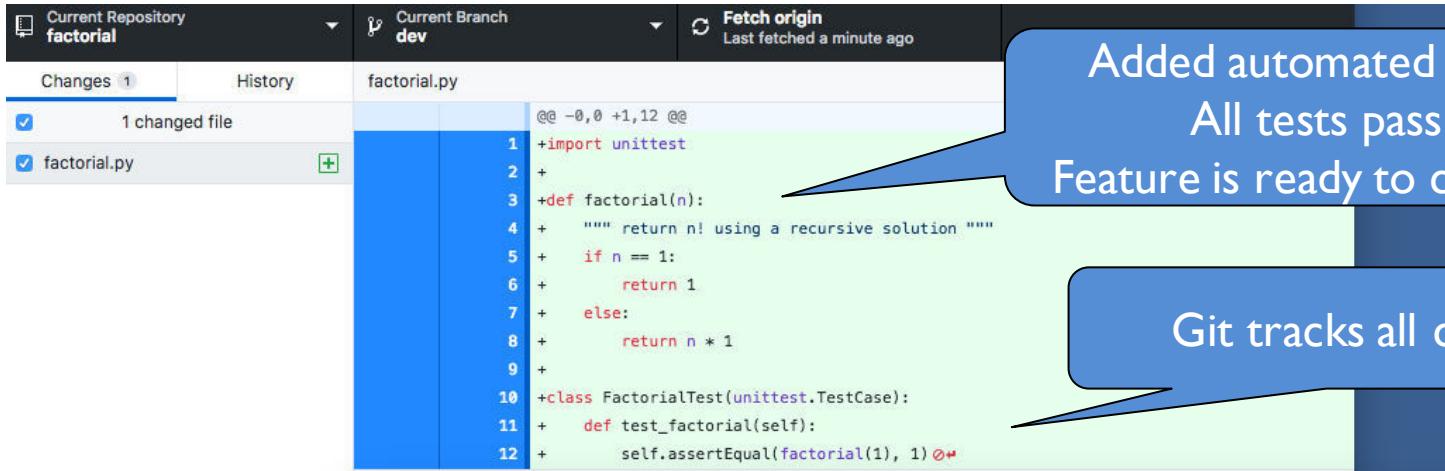
    new file:   factorial.py
jrr (dev +) factorial $
```

git status detects a  
new file in the  
Working Directory

Add the new file to the  
Staging Area, ready to commit

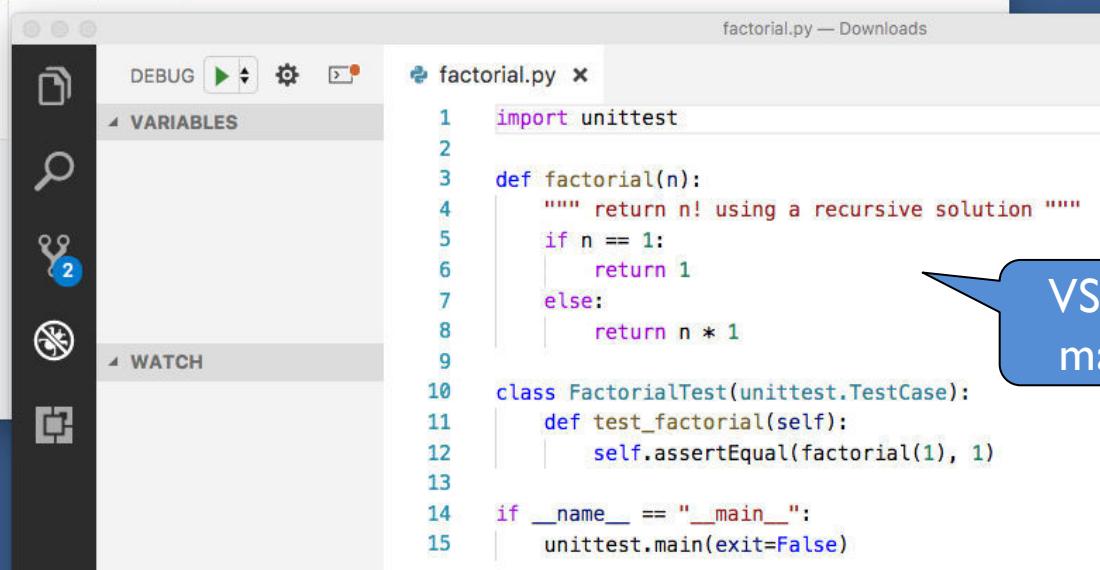
The new file has not  
been committed yet

# Continue changing the file



```

@@ -0,0 +1,12 @@
+import unittest
+
+def factorial(n):
+    """ return n! using a recursive solution """
+    if n == 1:
+        return 1
+    else:
+        return n * factorial(n-1)
+
+class FactorialTest(unittest.TestCase):
+    def test_factorial(self):
+        self.assertEqual(factorial(1), 1)
+
```



```

import unittest

def factorial(n):
    """ return n! using a recursive solution """
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

class FactorialTest(unittest.TestCase):
    def test_factorial(self):
        self.assertEqual(factorial(1), 1)

if __name__ == "__main__":
    unittest.main(exit=False)

```

VS Code, but you may use any tool

# Commit

A unit of work, i.e. a collection of related changes

May involve one or more files, but should represent a logical unit of functionality

Should always include a meaningful comment that is helpful to anyone reading the code

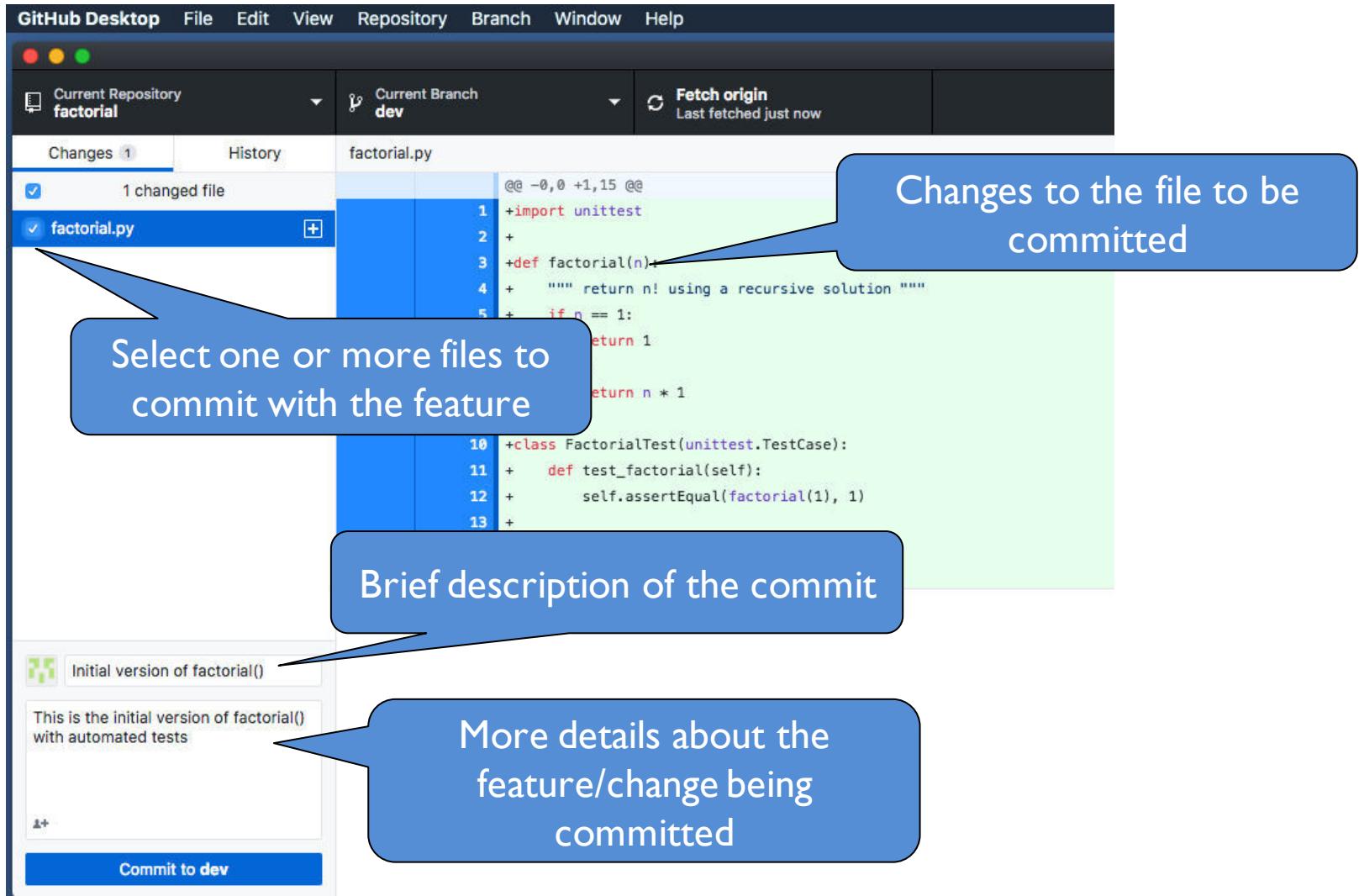
Changes should be committed frequently to a branch, but not until the feature is complete

Doing a commit may trigger continuous integration and automated testing



Travis CI

# Committing the change



The screenshot shows the GitHub Desktop application interface. At the top, the menu bar includes GitHub Desktop, File, Edit, View, Repository, Branch, Window, and Help. The title bar shows "Current Repository factorial" and "Current Branch dev". A status bar indicates "Fetch origin Last fetched just now".

The main area displays a "Changes" tab with one changed file, "factorial.py". A blue callout bubble points to the "factorial.py" file in the list, containing the following code:

```
@@ -0,0 +1,15 @@
+import unittest
+
+def factorial(n):
+    """ return n! using a recursive solution """
+
+    if n == 1:
+        return 1
+
+    return n * factorial(n-1)
+
+class FactorialTest(unittest.TestCase):
+    def test_factorial(self):
+        self.assertEqual(factorial(1), 1)
```

A large blue callout bubble contains the text: "Select one or more files to commit with the feature".

A blue callout bubble points to the code editor area, containing the text: "Changes to the file to be committed".

A blue callout bubble points to the bottom right of the code editor, containing the text: "Brief description of the commit".

In the bottom left corner, there is a commit message: "Initial version of factorial()", followed by the text: "This is the initial version of factorial() with automated tests".

A blue callout bubble points to this message, containing the text: "More details about the feature/change being committed".

At the bottom center, a blue button labeled "Commit to dev" is visible.



# Local commit doesn't change remote until push

The screenshot shows a GitHub repository page for 'Prof-JR / factorial'. The repository has 1 commit, 2 branches, 0 releases, and 1 contributor. It is currently on the 'dev' branch. A blue speech bubble points from the bottom left towards the repository stats, containing the text: 'The commit for factorial() to the local repository is not reflected in the remote repository yet...'. At the bottom, there is a call to action to 'Add a README'.

Search or jump to... / Pull requests Issues Marketplace Explore

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

No description, website, or topics provided. Edit

Add topics

1 commit 2 branches 0 releases 1 contributor

Branch: dev New pull request

This branch is even with master.

Prof-JR Initial commit .gitignore

Upload files Find file Clone or download ▾

Pull request Compare

Latest commit 49bd207 3 hours ago

3 hours ago

Help people interested in this repository understand your project by adding a README. Add a README

The commit for factorial() to the local repository is not reflected in the remote repository yet...



# git status after commit

```
jrr (dev) factorial $ git status
On branch dev
Your branch is ahead of 'origin/dev' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
jrr (dev) factorial $
```

The local dev branch is ahead of the remote repository by one commit. We need to choose when to update the remote repository.



# Push the local changes to the remote

The screenshot shows the GitHub Desktop application interface. The top bar includes 'GitHub Desktop' and other menu items like 'File', 'Edit', 'View', 'Repository', 'Branch', 'Window', and 'Help'. The main area displays a repository named 'factorial' with the current branch set to 'dev'. A blue callout box points from the text below to the 'Push origin' button in the top right corner of the main window. The status bar at the bottom indicates 'Push origin Last fetched a minute ago'.

Initial version of factorial()

Jim Rowland committed 7b07760

This is the initial version of factorial() with automated tests

factorial.py

```
@@ -0,0 +1,15 @@
1+import unittest
2+
3+def factorial(n):
4+    """ return n! using a recursive solution """
5+    if n == 1:
6+        return 1
7+    else:
8+        return n * 1
9+
10+class FactorialTest(unittest.TestCase):
11+    def test_factorial(self):
12+        self.assertEqual(factorial(1), 1)
13+
```

“Push origin” pushes the commits from the local repository to the remote repository and merges the changes

Good idea to check changes on remote before push



# The changes are now on the remote

Prof-JR / factorial

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

No description, website, or topics provided. Edit

Add topics

2 commits 2 branches 0 releases 1 contributor

Your recently pushed branches:

dev (3 minutes ago) Compare & pull request

Branch: dev New pull request

This branch is 1 commit ahead of master.

Prof-JR Initial version of factorial() ...

.gitignore Initial commit 4 hours ago

factorial.py Initial version of factorial() an hour ago

New factorial.py is now available to everyone from the remote repository on github.com

Latest commit 7b07760 an hour ago

Pull request Compare

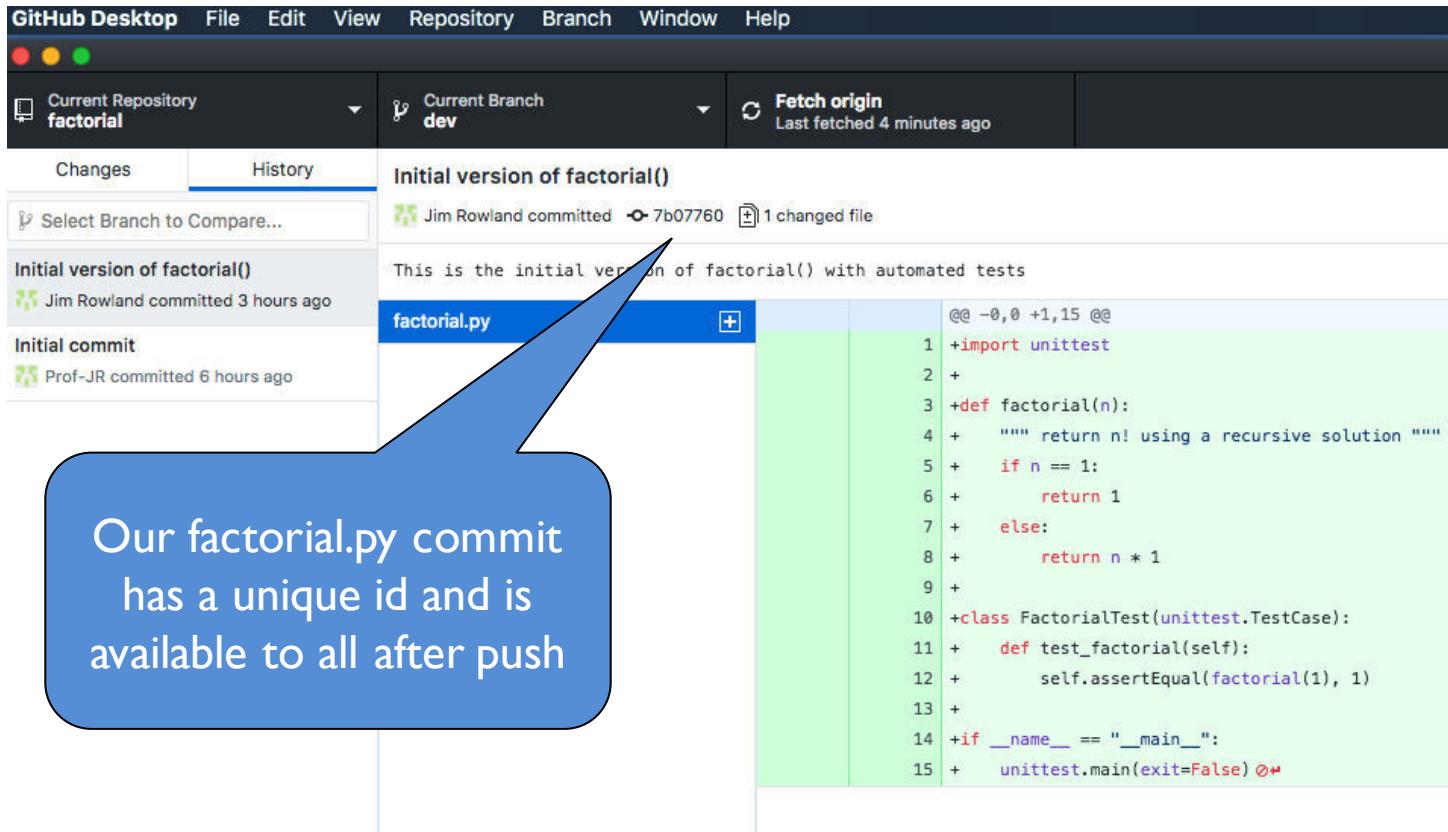
Add a README

A blue speech bubble points from the text "New factorial.py is now available to everyone from the remote repository on github.com" to the "Compare & pull request" button on the GitHub interface.

# Remote change to factorial.py

Files may be added or changed by other team members

Git helps to manage changes from many contributors



The screenshot shows the GitHub Desktop application interface. The top menu bar includes GitHub Desktop, File, Edit, View, Repository, Branch, Window, and Help. The main window displays a repository named "factorial" with a current branch of "dev". A "Fetch origin" status message indicates it was last fetched 4 minutes ago. On the left, a sidebar shows a commit history:

- Initial version of factorial()** (Jim Rowland committed 7b07760)
- Initial commit** (Prof-JR committed 6 hours ago)

A blue callout bubble points to the first commit with the text: "Our factorial.py commit has a unique id and is available to all after push". The right side of the screen shows a detailed view of the "factorial.py" file's initial version. The code is as follows:

```
@@ -0,0 +1,15 @@
1+import unittest
2+
3+def factorial(n):
4+    """ return n! using a recursive solution """
5+    if n == 1:
6+        return 1
7+    else:
8+        return n * factorial(n-1)
9+
10+class FactorialTest(unittest.TestCase):
11+    def test_factorial(self):
12+        self.assertEqual(factorial(1), 1)
13+
14+if __name__ == "__main__":
15+    unittest.main(exit=False) @@
```



# GitHub repository home

The repository page shows all directories and files in the project

Create, read, update files in the project from [github.com](https://github.com)

A screenshot of a GitHub repository page for 'Prof-JR / factorial'. The page includes a navigation bar with 'Code' selected, and sections for 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. A message 'No description, website, or topics provided.' is displayed, along with a 'Edit' button and a 'Add topics' link. Below this, summary statistics show '2 commits', '2 branches', '0 releases', and '1 contributor'. A dropdown menu shows 'Branch: dev' and a 'New pull request' button. A message 'This branch is 1 commit ahead of master.' is shown above a list of commits. The first commit by 'Prof-JR' is titled 'Initial version of factorial()' and was made 3 hours ago. Another commit for '.gitignore' was made 6 hours ago. A callout bubble points to the first commit with the text 'Click to see/edit a file from github.com'. At the bottom, a message encourages adding a README, with a 'Add a README' button.

Prof-JR / factorial

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

No description, website, or topics provided. Edit

Add topics

2 commits 2 branches 0 releases 1 contributor

Branch: dev New pull request

This branch is 1 commit ahead of master.

Prof-JR Initial version of factorial() 3 hours ago

.gitignore Initial commit 6 hours ago

factorial.py Initial version of factorial() 3 hours ago

Help people interested in this repository understand your project by adding a README. Add a README

Click to see/edit a file from github.com



# Edit files from github.com

Edit files and commit changes directly from github.com

Prof-JR / factorial

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

Branch: dev factorial / factorial.py Find file Copy path

Prof-JR Initial version of factorial() 7b07760 3 hours ago

1 contributor

15 lines (12 sloc) | 317 Bytes Raw Blame History

1 import unittest  
2  
3 def factorial(n):  
4 """ return n! using a recursive solution """  
5 if n == 1:  
6 return 1  
7 else:  
8 return n \* factorial(n - 1)  
9  
10 class FactorialTest(unittest.TestCase):  
11 def test\_factorial(self):  
12 self.assertEqual(factorial(1), 1)  
13  
14 if \_\_name\_\_ == "\_\_main\_\_":  
15 unittest.main(exit=False)

Need to fix the bug in factorial.py and add new test cases



# Edit files from github.com

factorial / factorial.py or cancel

Edit file Preview changes Spaces 4 No wrap

```
1 import unittest
2
3 def factorial(n):
4     """ return n! using a recursive solution """
5     if n == 1:
6         return 1
7     else:
8         return n * factorial(n - 1)
9
10 class FactorialTest(unittest.TestCase):
11     def test_factorial(self):
12         self.assertEqual(factorial(1), 1)
13         self.assertEqual(factorial(2), 2)
14         self.assertEqual(factorial(5), 120)
15
16 if __name__ == "__main__":
17     unittest.main(exit=False)
```

Commit changes

Bug fix for factorial n \* 1

Fixed problem with last line of factorial(n) to return n \* factorial(n -1).

Added new test cases

After changing the file on  
github.com, commit the change

Commit directly to the dev branch.

Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

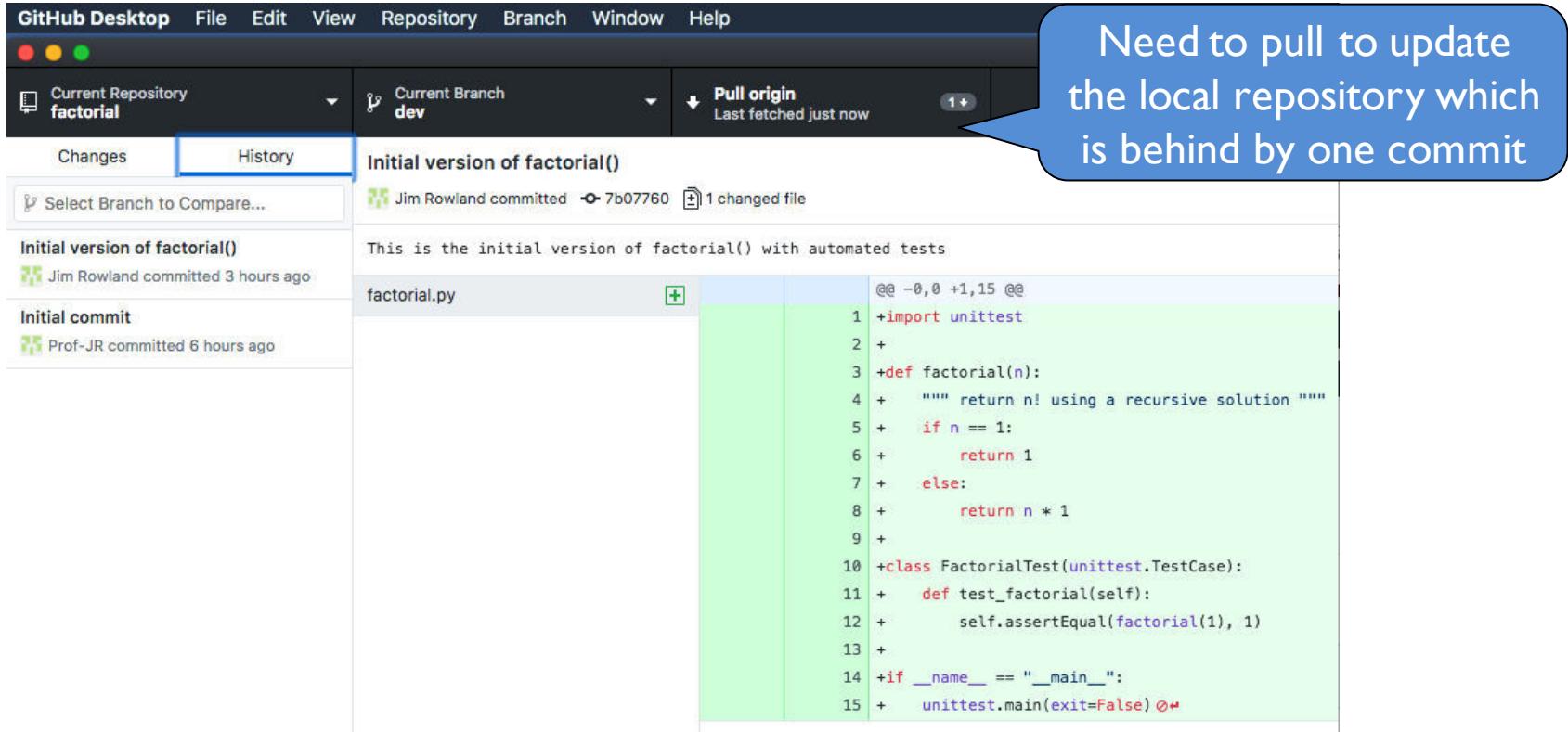
**Commit changes** Cancel

# Pull remote changes back to local

Someone updated factorial.py to fix problem

GitHub Desktop detected the change

Pull the changes from remote to update the local repository



The screenshot shows the GitHub Desktop application interface. At the top, the menu bar includes GitHub Desktop, File, Edit, View, Repository, Branch, Window, and Help. Below the menu is a toolbar with three colored dots (red, yellow, green). The main window has a dark header with "Current Repository factorial" and "Current Branch dev". A blue callout bubble points to the "Pull origin" button, which is highlighted with a blue border and contains the text "Last fetched just now". The main content area shows a commit titled "Initial version of factorial()". The commit details say "Jim Rowland committed 7b07760 1 changed file". The commit message is "This is the initial version of factorial() with automated tests". The diff view shows the contents of factorial.py. The code is as follows:

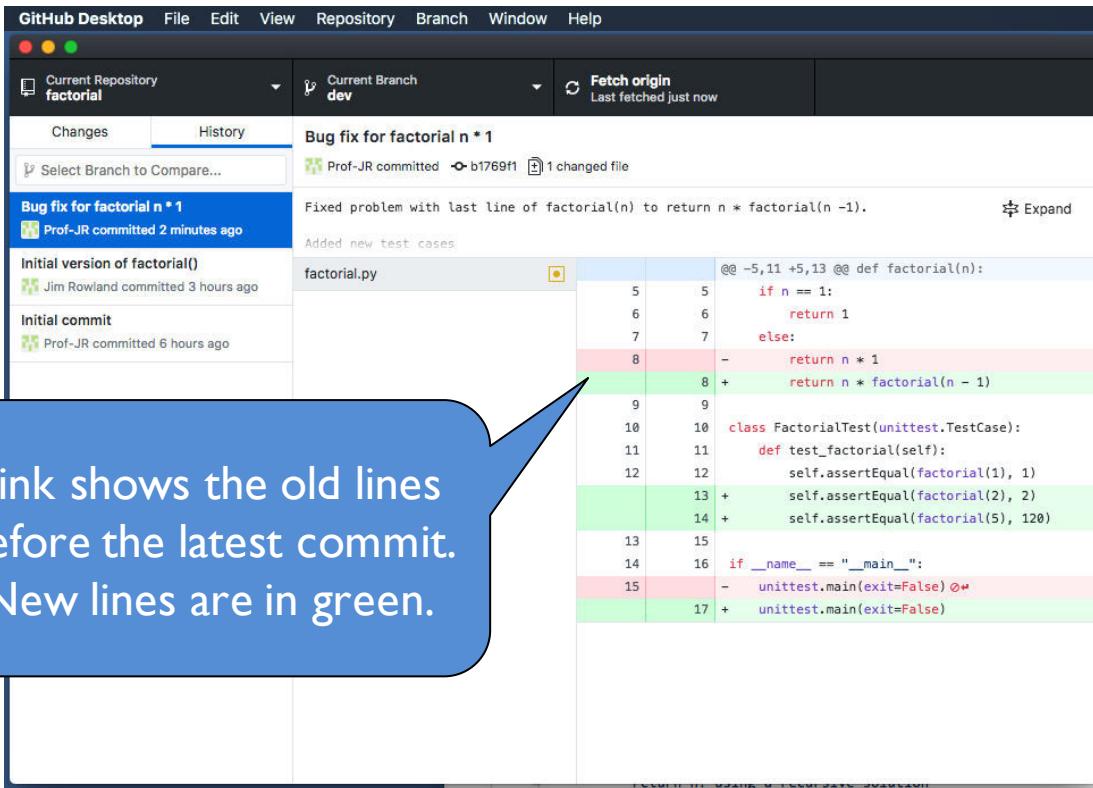
```
@@ -0,0 +1,15 @@
+import unittest
+
+def factorial(n):
+    """ return n! using a recursive solution """
+    if n == 1:
+        return 1
+    else:
+        return n * factorial(n-1)
+
+class FactorialTest(unittest.TestCase):
+    def test_factorial(self):
+        self.assertEqual(factorial(1), 1)
+
+if __name__ == "__main__":
+    unittest.main(exit=False)
```

Need to pull to update  
the local repository which  
is behind by one commit

# After pull, review the changes

Pulling from the remote updates the local repository to include the latest commits

Compare old and new versions to understand what changed



The screenshot shows the GitHub Desktop application interface. The top bar includes 'GitHub Desktop' and various menu options like File, Edit, View, Repository, Branch, Window, and Help. Below the menu is a toolbar with three colored dots (red, yellow, green). The main window displays a 'Current Repository' dropdown set to 'factorial', a 'Current Branch' dropdown set to 'dev', and a 'Fetch origin' section indicating 'Last fetched just now'. The central pane shows a commit history for the 'factorial' repository. The first commit is 'Bug fix for factorial n \* 1' by 'Prof-JR' committed 2 minutes ago. The second commit is 'Initial version of factorial()' by 'Jim Rowland' committed 3 hours ago. The third commit is 'Initial commit' by 'Prof-JR' committed 6 hours ago. A blue callout box points to the code diff for the 'Bug fix for factorial n \* 1' commit. The diff highlights changes in 'factorial.py'. Pink highlights show the old lines before the latest commit, and green highlights show the new lines added. The code snippet is as follows:

```
factorial.py
@@ -5,11 +5,13 @@ def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
+
class FactorialTest(unittest.TestCase):
    def test_factorial(self):
        self.assertEqual(factorial(1), 1)
        self.assertEqual(factorial(2), 2)
        self.assertEqual(factorial(5), 120)
+
if __name__ == "__main__":
    unittest.main(exit=False)
+
unittest.main(exit=False)
```

Pink shows the old lines  
before the latest commit.  
New lines are in green.

# Pull requests

GitHub supports geographically dispersed teams working on open source projects

Commits may be made by potentially “untrusted” sources

A pull request informs the project leader that changes are available for review and encourages discussion about the changes

Project leader pulls all relevant changes together and chooses which changes to merge into a current or new branch





# New Pull Request

Initiate a pull request to pull the changes from ‘dev’ into ‘master’  
You should be sure to test your system before merging the pull request into the Master Branch

The screenshot shows a GitHub repository page for 'Prof-JR / factorial'. The page includes a navigation bar with 'Watch' (0), 'Star' (0), and 'Fork' (0) buttons. Below the bar are links for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. A note says 'No description, website, or topics provided.' with an 'Edit' button. A 'Add topics' link is also present. The main area shows '3 commits' and a large blue speech bubble pointing to the 'New pull request' button. Below the commits, it says 'This branch is 2 commits ahead of master.' with 'Pull request' and 'Compare' buttons. A list of commits includes: 'Prof-JR Bug fix for factorial n \* 1' (latest commit b1769f1 17 hours ago), '.gitignore' (Initial commit 23 hours ago), and 'factorial.py' (Bug fix for factorial n \* 1 17 hours ago). At the bottom, there's a note to 'Help people interested in this repository understand your project by adding a README.' with a 'Add a README' button.



# New Pull Request

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

The screenshot shows a GitHub pull request interface. At the top, it says "base: master" and "compare: dev". A green checkmark indicates "Able to merge". The main area has a title "Initial release of factorial" and a description "factorial(n) written, tested, and optimized." Below this is a text input field with placeholder "Attach files by dragging & dropping, selecting them, or pasting from the clipboard." A "Create pull request" button is at the bottom right. The commit history shows two commits from "Prof-JR": "Initial version of factorial()" and "Bug fix for factorial n \* 1". The file changes section shows "Showing 1 changed file with 17 additions and 0 deletions" and a diff view for "factorial.py". Three blue speech bubbles with white text are overlaid on the interface:

- A large blue bubble points to the "factorial(n) written, tested, and optimized." text area with the text "Document the features".
- A medium blue bubble points to the "Create pull request" button with the text "Click when ready".
- A smaller blue bubble points to the "Review changes" section of the file diff with the text "Review changes".

```
17  factorial.py
...
1  +import unittest
2  +
3  +def factorial(n):
4  +    """ return n! using a recursive solution """
5  +    if n == 1:
```



# New Pull Request

## Initial release of factorial #1

[Open](#) Prof-JR wants to merge 2 commits into `master` from `dev`

Conversation 0 Commits 2 Checks 0 Files changed 1 +17 -0

Prof-JR commented a minute ago  
factorial(n) written, tested, and optimized.

Prof-JR added some commits 21 hours ago

- Initial version of factorial()
- Bug fix for factorial n \* 1

Review the commits in the release

Add more commits by pushing to the `dev` branch on Prof-JR/factorial.

Continuous integration has not been set up  
Several apps are available to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch  
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Merge changes from 'dev' into 'master'

Continuous Integration causes automated tests to run on every commit – **highly recommended!**

Write Preview Leave a comment Attach files by dragging & dropping, selecting them, or pasting from the clipboard. Styling with Markdown is supported Close pull request Comment

1 participant Lock conversation



# Pull requests

Pull requests allow everyone on the team to review changes before the changes are merged into the Master Branch

You are asking the team to pull your changes into the Master Branch

GitHub provides a collaborative environment to allow teams to work together in the same room or geographically dispersed.





# Review...

Walk through a common scenario...

1. Create a remote repository for a new project on github.com
2. Clone the remote repository to a local repository
3. Create a new ‘dev’ branch
4. Create a new file in the local repository
5. Commit changes locally
6. Push changes to the remote repository
7. Change the file in the remote repository
8. Pull the changes from the remote repository
9. Merge ‘dev’ and ‘master’ branch



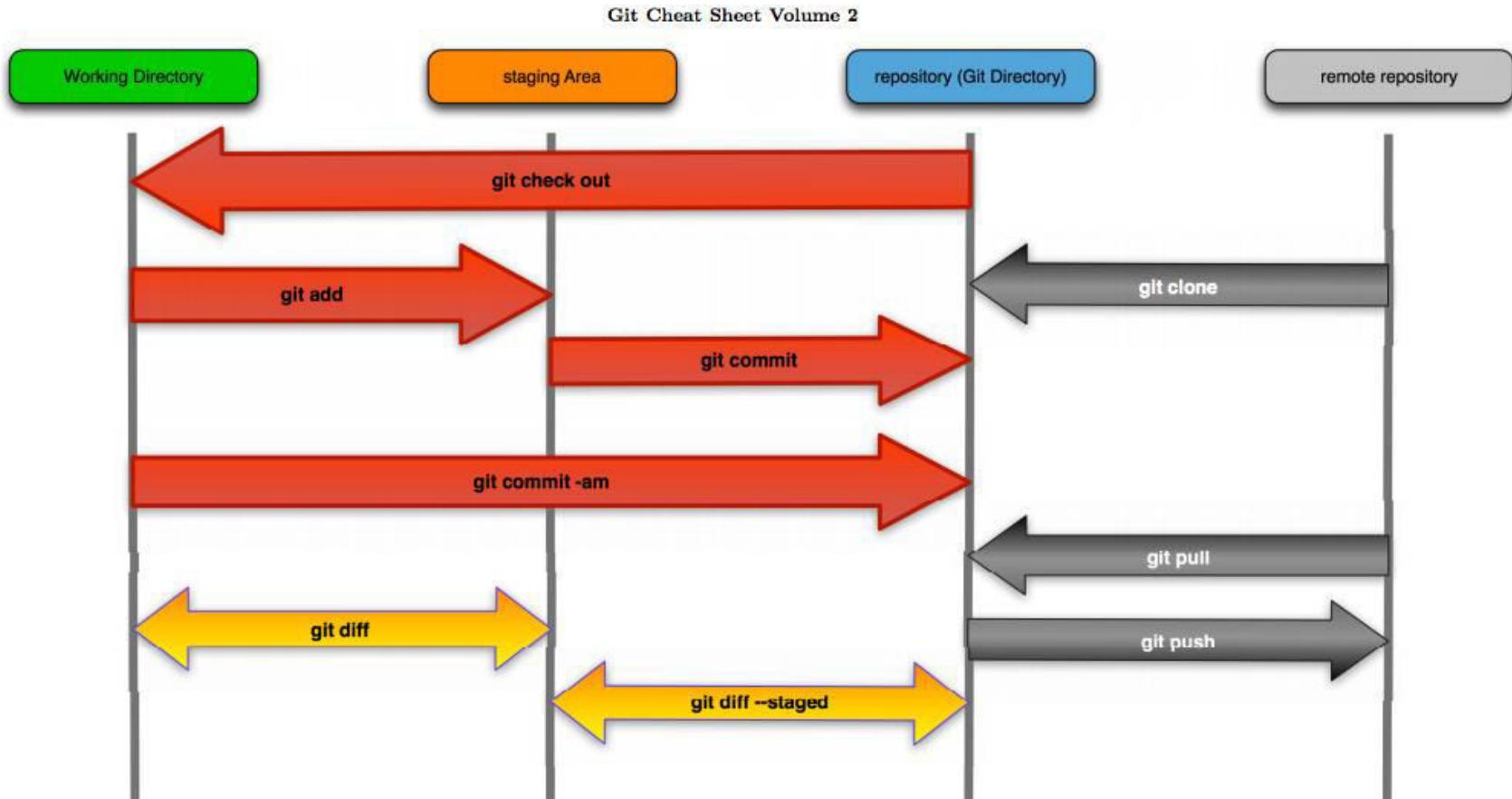
© Can Stock Photo



# git commands

init	• Initialize a new repository
clone	• Clone an existing repository
add	• Mark a file ready for commit
commit	• Commit changes to the repository
status*	• See the status of all files in the repository
diff*	• Compare working directory to repository
checkout	• Switch to a different branch
branch	• Create a new branch
tag	• Add a tag/label to a branch
log	• See changes made to the repository
fetch	• retrieve remote changes, but don't change local
pull	• retrieve and <b>merge</b> remote changes locally
push	• Push a repository to remote server

# git Cheat Sheet




---

Date: February 20, 2013 - Version 1.1  
 Author: Max Oberberger ([github@oberbergers.de](mailto:github@oberbergers.de))  
<https://github.com/chiemseesurfer/latex-gitCheatSheet>

# More GitHub features

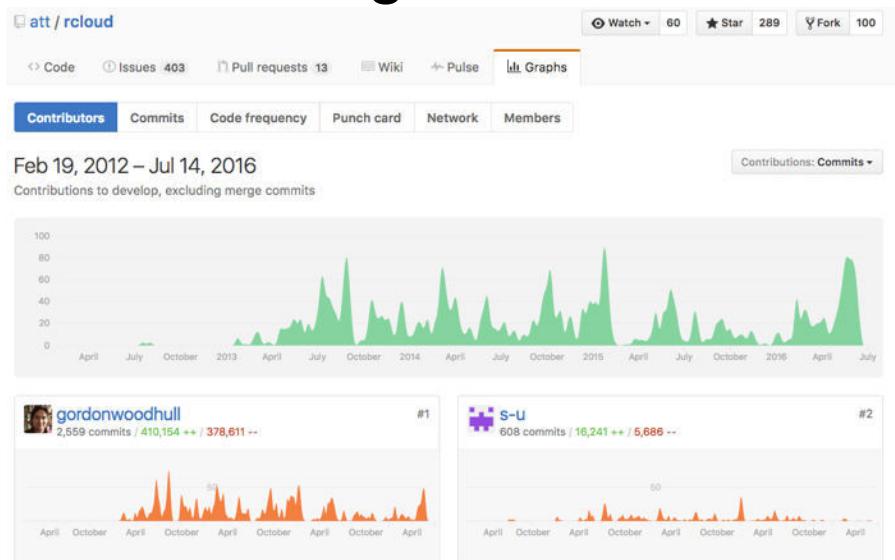
GitHub offers other useful features

Issue tracking – Create and track project features and issues

Wiki – Describe and discuss the project

Pulse – Show recent activity

Graphs – See who's doing what





# Issues

att / rcloud

Watch ▾ 60

Star 289

Fork 100

Code

Issues 403

Pull requests 13

Wiki

Pulse

Graphs

Filters ▾

is:issue is:open

Labels

Milestones

New issue

403 Open ✓ 1,414 Closed

Author ▾

Labels ▾

Milestones ▾

Assignee ▾

Sort ▾

Clicking 'cancel' on 'New Asset' prompt throws error

#2171 opened 6 days ago by shaneporter ↗ 1.6

1

htmlwidget DiagrammeR doesn't work bug

#2169 opened 7 days ago by gordonwoodhull ↗ 1.6.1



5

Notebooks I Starred should always be present in All Notebooks, even if owner deleted them bug

#2165 opened 7 days ago by amithrb ↗ 1.6.1



2

starred hidden/deleted/unacknowledged notebooks of other users don't show up in All Notebooks

bug

#2160 opened 13 days ago by gordonwoodhull ↗ 1.6.1

2

discover page show times enhancement

#2150 opened 15 days ago by gordonwoodhull ↗ 1.6.1



7



# Pull Requests

att / rcloud

Watch 60 Star 289 Fork 100

Code Issues 403 Pull requests 13 Wiki Pulse Graphs

Filters ▾ is:pr is:open Labels Milestones New pull request

13 Open ✓ 344 Closed Author ▾ Labels ▾ Milestones ▾ Assignee ▾ Sort ▾

Paste or drop thumbnail dialog	#2175 opened 5 hours ago by shaneporter	4 comments
Docker	#2174 opened 6 days ago by prateek05	1.6.1 fix 4 comments
Display relative dates/times on Discover	#2170 opened 6 days ago by shaneporter	1.6.1 fix 1 comment
Filter rcloud extensions, take 2	enhancement #1609 opened on Jul 29, 2015 by gordonwoodhull	1.6.1 fix 3 comments
fiddleR	enhancement #1379 opened on Mar 4, 2015 by bartbutler	1.6.1 fix 9 comments



# Wiki

att / rcloud

Watch 60 Star 289 Fork 100

Code Issues 403 Pull requests 13 Wiki Pulse Graphs

## Home

Simon Urbanek edited this page on Mar 18 · 18 revisions

Edit New Page

## Configuration

- Installation instructions are in [INSTALL.md](#).
- The main configuration file is [conf/rcloud.conf](#)

## Need help?

You can [file a Github issue](#).

## User documentation

- [notebook.R-API](#)
- [mini.html](#) minimalistic API for webpages to use RCloud notebooks in sessions

## Developer documentation

- [RCS](#) key/value store in RCloud, conventions and considerations
- [RCloud Extensions](#) add languages and user interface elements to RCloud
- R and JavaScript functions call each other using [R-JavaScript binding](#)

## Other

[Similar Tools](#)

▼ Pages (18)

Find a Page...

Home  
Enhancement TO DO List  
mini.html  
notebook.R API  
PrivateNotebooks  
ProxifiedSetup  
R JavaScript binding  
RCloud Extensions  
RCloud Language extensions  
RCloud RPC internals  
RCloud UI Extensions  
rcloud.conf  
RCS  
Release steps  
Required Installations for running Python code in RCloud  
Show 3 more pages...



# Pulse

att / rcloud

Watch 60 Star 289 Fork 100

Code Issues 403 Pull requests 13 Wiki Pulse Graphs

Period: 1 week ▾

July 7, 2016 – July 14, 2016

Overview

4 Active Pull Requests	4 Active Issues		
Merged Pull Request 1	Proposed Pull Requests 3	Closed Issues 2	New Issues 2

Excluding merges, 3 authors have pushed 7 commits to develop and 7 commits to all branches. On develop, 7 files have changed and there have been 20 additions and 18 deletions.

1 Pull request merged by 1 person

Merged #2172 Bump rcloud.support version to 1.6 6 days ago

3 Pull requests proposed by 2 people

Proposed #2170 Display relative dates/times on Discover 6 days ago

Proposed #2174 Docker 6 days ago

Proposed #2175 Paste or drop thumbnail dialog 5 hours ago



# Graphs

att / rcloud

Watch ▾ 60 Star 289 Fork 100

Code Issues 403 Pull requests 13 Wiki Pulse

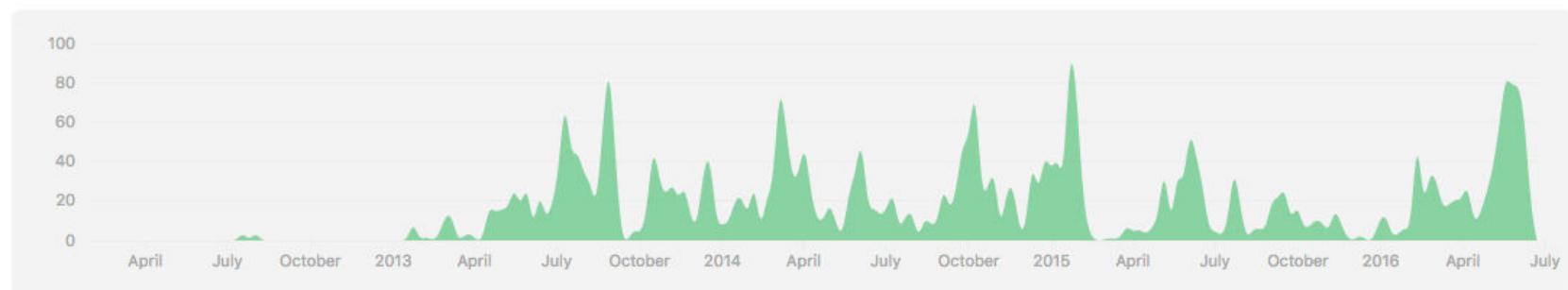
Graphs

Contributors Commits Code frequency Punch card Network Members

Feb 19, 2012 – Jul 14, 2016

Contributions: Commits ▾

Contributions to develop, excluding merge commits



# What to store in GitHub?

All important project artifacts

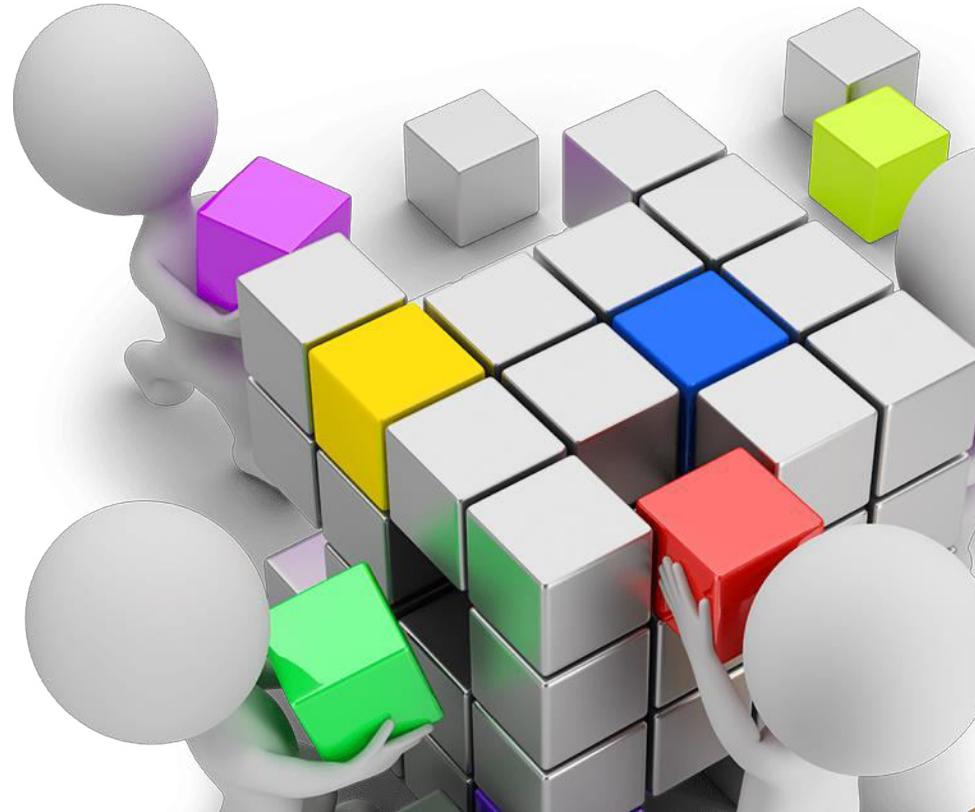
Source files

Documents

Test cases

Test results

...





# Resources

## Pro Git : Chacon and Straub

<https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf>

## Getting started with GitHub

<https://guides.github.com/activities/hello-world/>

<https://guides.github.com/>

<https://www.git-tower.com/learn/>

# To Do

Register for a free GitHub account at [www.github.com](http://www.github.com)

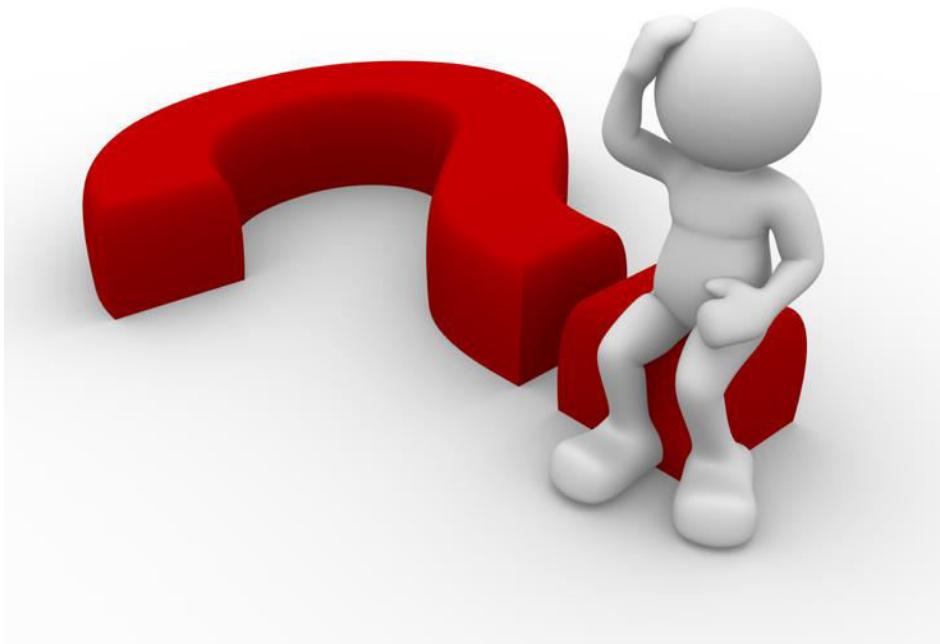
Work through the README tutorial at  
<https://guides.github.com/activities/hello-world/>

Start using git or GitHub for your homework assignments

The remainder of the assignments are related and build on the assignments from earlier weeks



# Questions?





# SSW-810: Software Engineering Tools and Techniques

## *Relational Database Technologies and Techniques*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from:

<http://www.tutorialspoint.com/dbms/>

[https://www.tutorialspoint.com/sqlite/sqlite\\_python.htm](https://www.tutorialspoint.com/sqlite/sqlite_python.htm)

# Today's topics

Database overview

Common database solutions

Have you seen my keys?

Queries

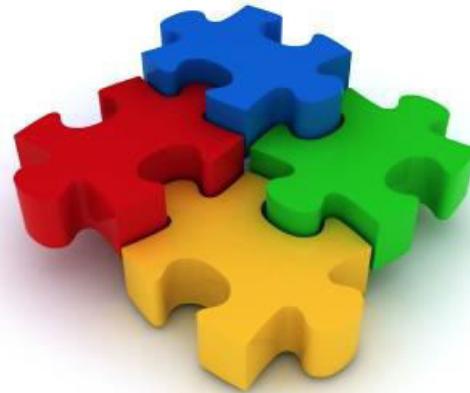
Select

Update

Delete

Defining tables

Interacting with databases from Python





# Database Overview

Databases contain a collection of related information, organized to allow efficient access and retrieval

Several different database solutions:



**Relational Database Management System (RDBMS)** – data is structured in predefined tables by row and column

**Object Oriented Databases** – all data is represented and stored by objects

**NoSQL** - unstructured data is stored and processed in potentially massively parallel solutions



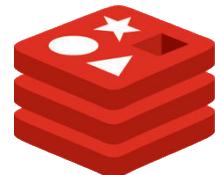
# Many Database Solutions

ORACLE®

MySQL™

MariaDB

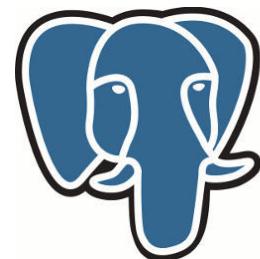
SQLite



redis

mongoDB.  
FOR GIANT IDEAS

Microsoft®  
SQL Server®



PostgreSQL  
the world's most advanced open source database

# Why use a RDBMS?

Why not just store data in a flat file?

(Flat files may be the best solution for some problems)

Databases support:

- Transaction processing/Atomicity
- Multiple concurrent simultaneous users
- A common query language across vendors
- Data consistency enforcement
- Access control



# ACID properties



A ***transaction*** must have **ACID** properties

**Atomic** - all steps complete successfully or no change

**Consistent** – both successful and unsuccessful transactions must leave the system in a consistent state

**Isolated** – the data involved in the transaction must be isolated from other users until the transaction is complete

**Durable** – the changes must survive permanently

# Transaction Atomicity

Consider a banking application where a customer transfers money from one account to another

Transfer requires two steps:

1. Deduct funds from first account
2. Deposit funds to second account



What if something goes wrong in either step?

Need to ensure that **both** steps are successful or the account can be left in inconsistent state

Complete **both** steps successfully or rollback **all** changes

# Relational DBMS Characteristics

Data represents real world entities

Data stored in relational tables

Tables are analogous to spreadsheets with rows and columns

Tables may be related by “keys” which can be used to join multiple tables to find related information

Tables are designed to minimize redundant information

Use query languages (SQL) to retrieve information

Support transactions for consistency

Support multiple concurrent users



# Example Domain

Create a database solution

Start up company has employees and projects

Track who's working on which projects



# Entity Relationship Model

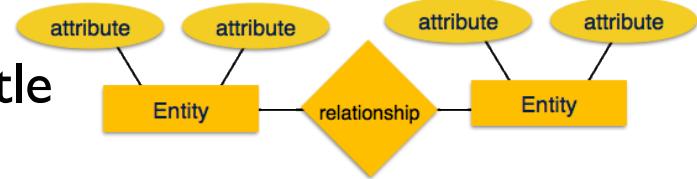
**Entities** represent some physical or logical object

E.g. employee, manager, project ...

Entities may have **attributes**

An employee has an ID number and a title

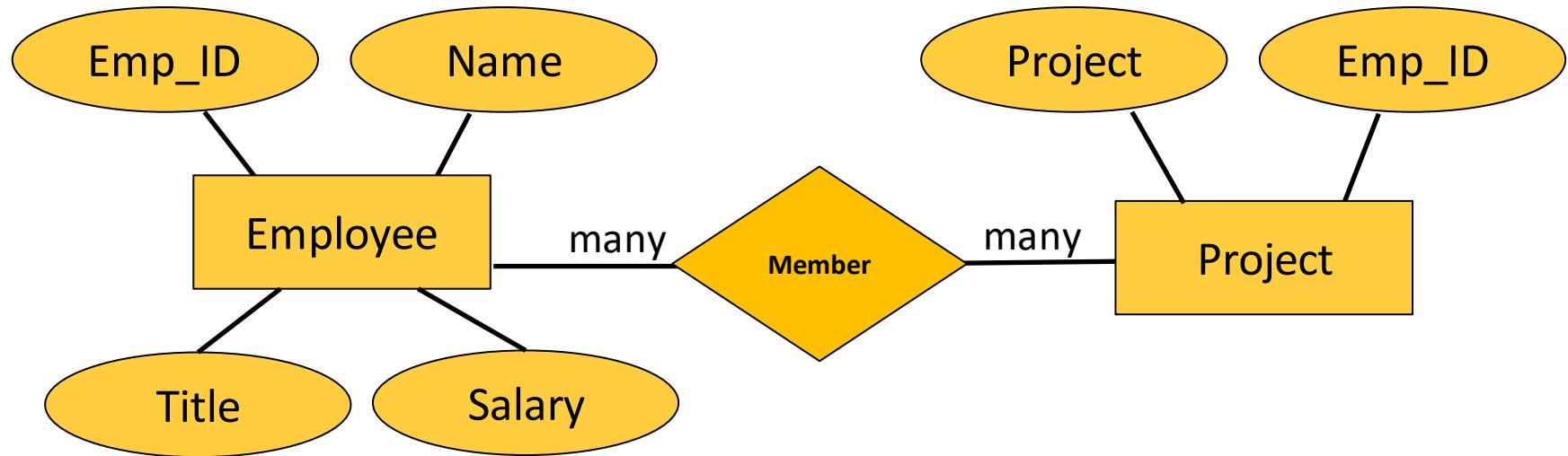
A project may have a set of employees



**Keys** are attributes that uniquely identify one instance of an entity from other instances

e.g. an employee ID uniquely identifies each employee

# Entity Relationship Models



**Employees Table**

Emp_ID	Name	Title	Salary
123	Nanda	Developer	80000
234	Sujit	Tester	70000
345	Jim	Manager	85000

**Projects Table**

Project	Emp_ID
OnlineBanking	123
GameConsole	234
OnlineBanking	234



# Representing Entities in a Relational DB

**Entities** are typically represented in **tables**

**Attributes** are represented as **columns** in the table

Instances are represented as **tuples** (rows) in the table

Tables **may** include a primary key to uniquely identify the instance of the entity represented by the tuple/row

Employees Table

Emp_ID (Key)	Name	Title	Salary
123	Nanda	Developer	80000
234	Sujit	Tester	70000
345	Jim	Manager	85000

Projects Table

Project	Emp_ID
OnlineBanking	123
GameConsole	234
OnlineBanking	234

# Database Tools

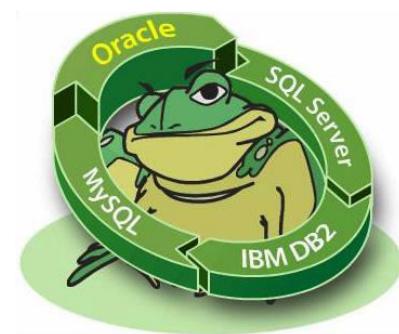
Many different tools to work with databases

Create and explore tables

Write and run queries



DataGrip





# Database operations

Command	Semantics
<b>Create</b>	Create a database, table, or view
<b>Drop</b>	Drop a database, table, or view
<b>Alter</b>	Modify the database schema
<b>Insert</b>	Insert new records into the table
<b>Update</b>	Update one or more attributes in an existing record
<b>Delete</b>	Delete one or more rows from a table
<b>Select</b>	Select a subset of records from the table based on selection criteria



SQLite is a small, open source, file-based RDBMS

Download from <https://sqlite.org/download.html>

Choose the precompiled binaries for your laptop

Works with files or in memory databases

Create a new database

```
$ cd 810/assignments
```

```
$ sqlite3
```

Connected to a **transient in-memory database**.

```
sqlite> .save 810_startup.db
```

```
sqlite> .exit
```

SQLite supports in-memory  
and on disk databases

Create a SQLite database  
file in 810\_startup.db

Or create database in DataGrip



# DataGrip Features

Connect to many different relational databases

SQLite, Oracle, MySql, PostgreSQL...

GUI automates many common tasks

Create a new database file

Table definition

Create data table from file

Save query output to a text file

Auto completion for table names and attributes

Free download for academic use

Part of the JetBrains product line, including PyCharm



**DataGrip**



# JetBrains DataGrip

The screenshot shows the JetBrains DataGrip interface for managing SQLite databases. The left sidebar displays the database structure, including tables like Employees and Projects with their respective columns. The main pane shows a SQL query editor with several queries written in SQL. Below the editor is a results viewer showing the output of a query. The bottom right corner features an event log tab showing database activity and errors.

Explore table definitions

Write and execute queries

See query outputs

General information

```
Sqlite (Xerial) - 810_startup.db.sql - StudentDB
```

```
1 select p.project, e.Title  
2 from Employees e join Projects p on e.Emp_ID=p.Emp_ID where e.Title='Architect'  
3  
4 select * from Employees  
5  
6 select Emp_ID, P  
7  
8 select sum(Salary)  
9  
10  
11  
12 select Name, Title, Salary from Employees order by Salary DESC  
13  
14 select count(distinct(Project)) from Projects
```

Project	Title
1 OnlineBanking	Architect
2 GameConsole	Architect

```
Event Log: General Database  
8:19 PM Select avg_(Salary) from Employees  
8:19 PM [1] [SQLITE_ERROR] SQL error or missing database (near "desc": syntax error)  
8:22 PM select Title, count(*) as cnt, av  
8:22 PM [1] [SQLITE_ERROR] SQL error or missing database (near "order": syntax error)  
select Name, Title from Employees  
group by Salary desc  
[1] [SQLITE_ERROR] SQL error or missing database (near "desc": syntax error)  
order by Salary desc  
[1] [SQLITE_ERROR] SQL error or missing database (near "order": syntax error)
```

[1] [SQLITE\_ERROR] SQL error or missing database (near "order": syntax error) (yesterday 8:25 PM)

106 chars, 2 lines 1:1 LF: UTF-8:



# Create a new database in DataGrip

The screenshot shows the DataGrip interface with a blue callout pointing to the 'SQLite' option in the 'Data Source' list. The 'Database' tab is selected in the sidebar. The main panel displays a table titled 'main.Employees [810\_startup.db]' with columns 'Name', 'Role', and 'Salary'. The table contains five rows of data:

Name	Role	Salary
JIM	Manager	80000
Fei	Architect	85000
Siddarth	DBA	90000
Sujit	Tester	87000
		70000

1. Choose 'New'

Choose 'SQLite'

Database

File Edit Database Tools Help

Console [StudentDB19F.sqlite] main.Employees [810\_startup.db] main.majors [StudentDB19F.sqlite]

Tx: Auto DB DDL

Data Source SQLite

- DDL Data Source
- Data Source from URL
- Data Source from Path
- Driver and Data Source
- Driver
  - MySQL
  - Oracle
  - PostgreSQL
  - Microsoft SQL Server
  - SQLite
  - Hive
  - Apache Derby
  - Apache Cassandra
  - Amazon Redshift
  - Greenplum
  - Exasol
  - ClickHouse
  - H2
  - IBM Db2
  - MariaDB
  - MySQL
  - Oracle
  - PostgreSQL
  - Snowflake
  - Sybase ASE
  - Vertica

WWD-Hannah.sqlite WWD-test.sqlite



# Create a new database in DataGrip

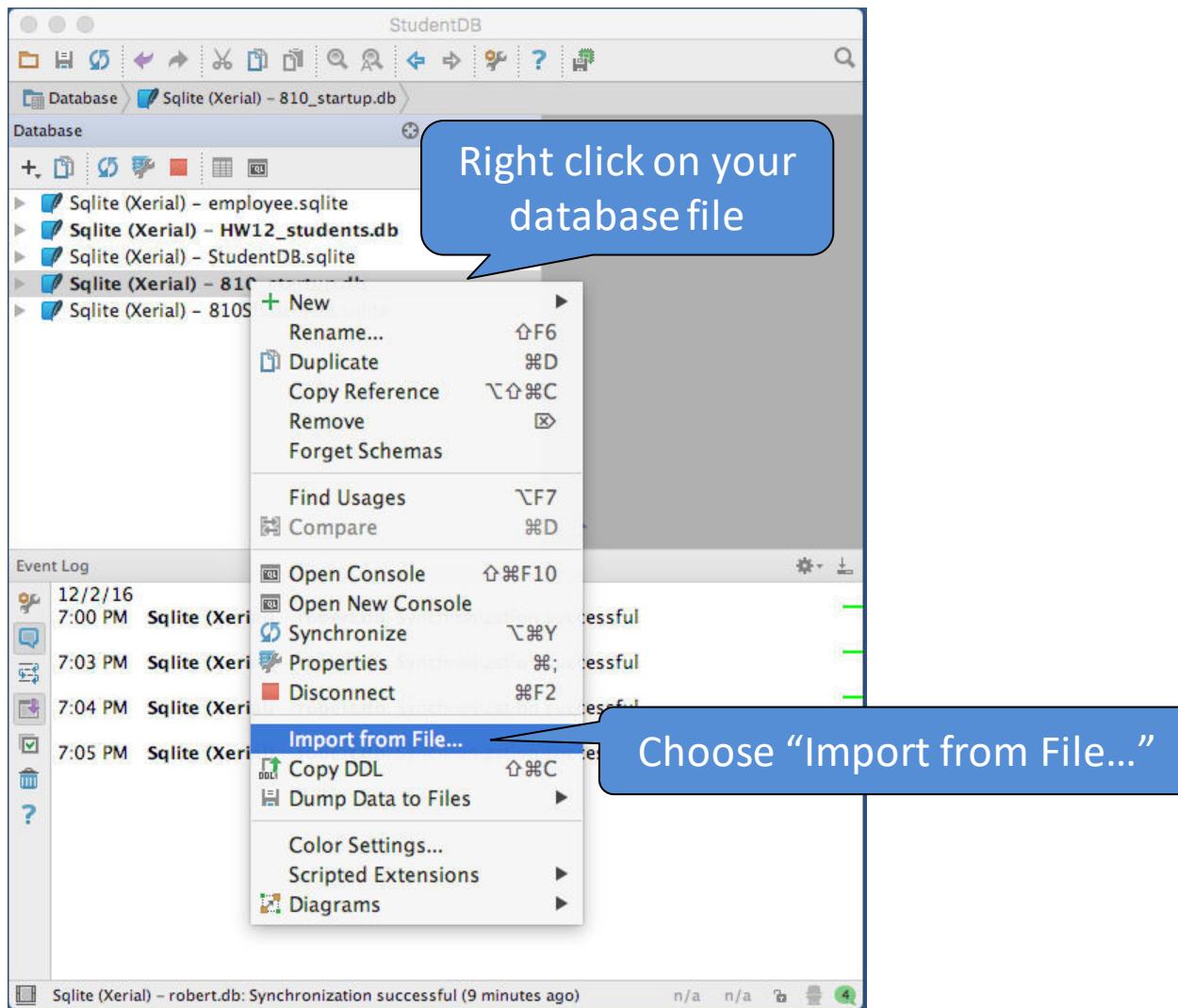
The screenshot shows the DataGrip interface with the following details:

- Left Sidebar:** Shows a tree view of databases and schemas. Databases listed include 17Y, database.db, movies.db, phd2001.db, project.db, project.db [2], project3.db, student\_assistant.db, StudentDB19F.sqlite, WWD-Hannah.sqlite, WWD\_test.sqlite, and x810\_startup.db. Under x810\_startup.db, there are schemas main (Employees, HW12\_students, Projects, sqlite\_master) and collations 3.
- Central Area:** A modal dialog titled "Data Sources and Drivers" is open, specifically for the "Project Data Sources" tab. It shows the "810\_startup.db" entry selected.
- Configuration Fields:**
  - Name: 810\_startup.db
  - Comment: (empty)
  - General tab (selected): Connection type: default, Driver: SQLite
  - File: 810\_startup.db
  - URL: jdbc:sqlite:810\_startup.db
  - Test Connection button (disabled)
  - Details:
    - DBMS: SQLite (ver. 3.25.1)
    - Case sensitivity: plain=mixed\_delim
    - Driver: SQLite JDBC (Ping: 12 ms)
- Bottom Buttons:** Cancel, Apply, OK

**Annotations:**

1. Specify a directory and filename (points to the "Name" field in the modal).
2. Test the new database connection (points to the "Test Connection" button in the modal).

# DataGrip: Import a file into a table





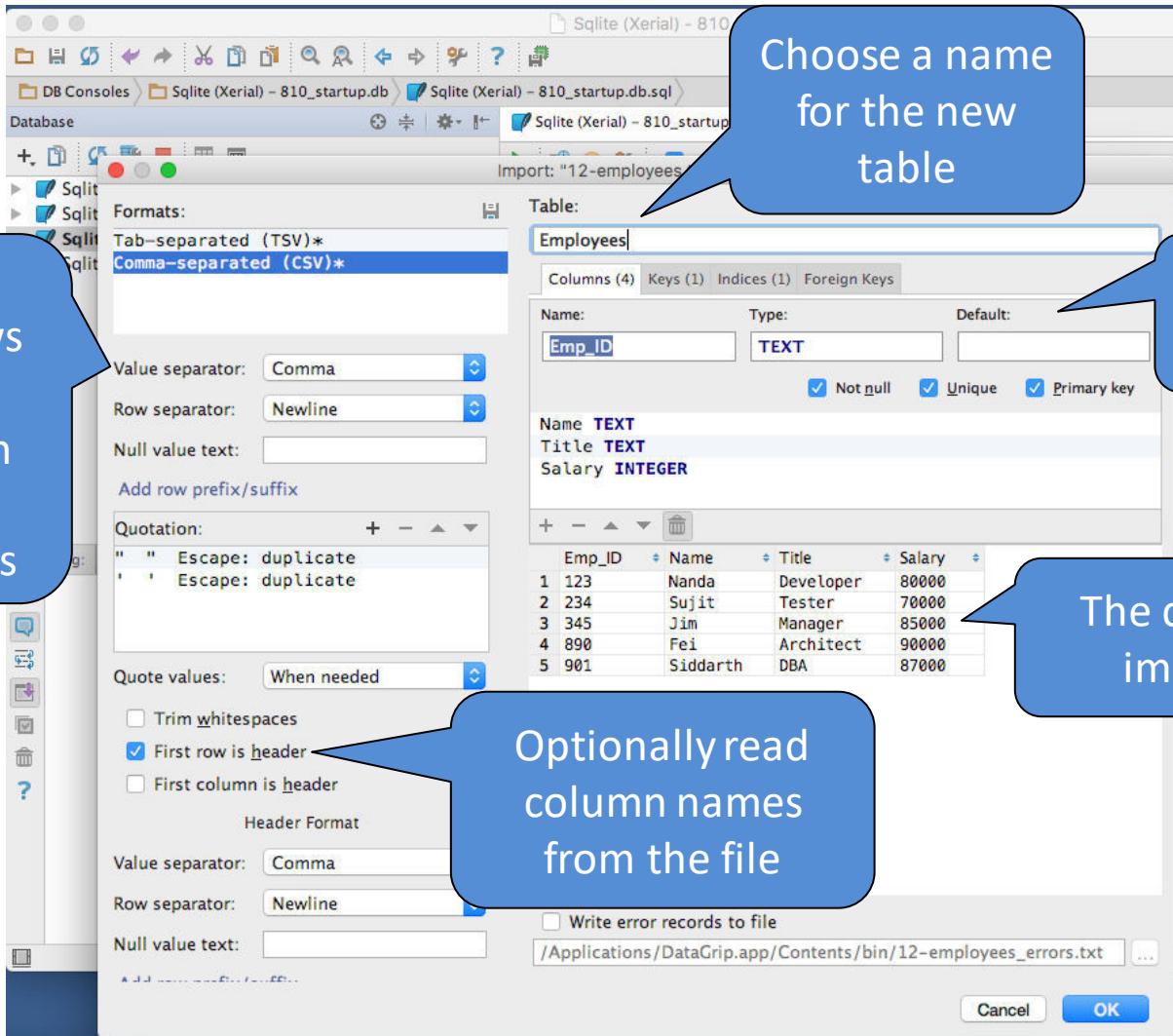
# Create a new table

Specify:

- The table name
- Each attribute in the table
- The data type of each attribute in the table:
  - INTEGER, REAL, TEXT, BLOB
- Optional constraints on attributes
  - NOT NULL
- PRIMARY KEY
- FOREIGN KEY

DBMS may support  
different types

# Create a new table from a data file



The screenshot shows the DataGrip interface with the 'Import' dialog open. The dialog is titled 'Import: "12-employees"' and is set to 'Comma-separated (CSV)\*'. The 'Table' field is set to 'Employees'. The 'Columns' tab shows four columns: 'Emp\_ID' (Type: TEXT, Primary key), 'Name' (Type: TEXT), 'Title' (Type: TEXT), and 'Salary' (Type: INTEGER). Below the table definition is a preview grid with the following data:

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

On the left side of the dialog, there are several configuration options:

- Formats:** Tab-separated (TSV)\* (disabled) and Comma-separated (CSV)\* (selected).
- Value separator:** Comma.
- Row separator:** Newline.
- Null value text:** [empty field].
- Add row prefix/suffix:** Quotation: "", Escape: duplicate; ', ' Escape: duplicate.
- Quote values:** When needed.
- Header Format:** Value separator: Comma, Row separator: Newline, Null value text: [empty field].
- Optional checkboxes:** Trim whitespaces (unchecked), First row is header (checked), First column is header (unchecked).
- Write error records to file:** /Applications/DataGrip.app/Contents/bin/12-employees\_errors.txt.

Blue callout boxes highlight specific features:

- A box points to the 'Table' field with the text: 'Choose a name for the new table'.
- A box points to the 'First row is header' checkbox with the text: 'Adjust name and type'.
- A box points to the preview grid with the text: 'The data to import'.
- A box points to the 'First row is header' checkbox with the text: 'Optional read column names from the file'.

DataGrip only allows comma, semicolon or tab separators

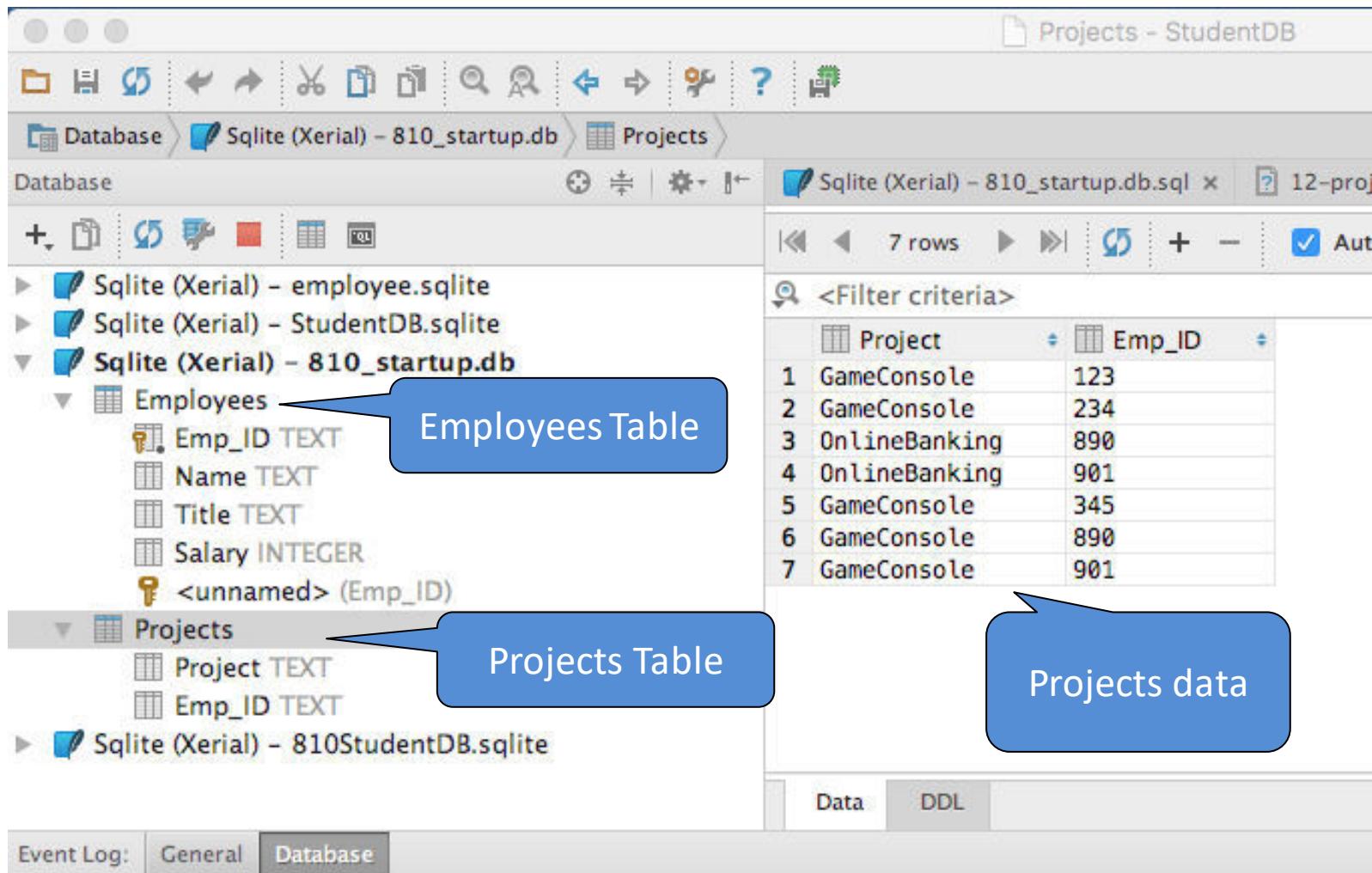
Choose a name for the new table

Adjust name and type

The data to import

Optional read column names from the file

# Repeat for Projects Table



The screenshot shows the SQLite Database Browser interface. On the left, the database tree displays several SQLite databases. A blue callout points to the "Employees" table under the "Sqlite (Xerial) - 810\_startup.db" database, which is highlighted. Another blue callout points to the "Projects" table under the same database. The main pane shows the content of the "Projects" table, with a third blue callout pointing to its data. The table has two columns: "Project" and "Emp\_ID". The data is as follows:

	Project	Emp_ID
1	GameConsole	123
2	GameConsole	234
3	OnlineBanking	890
4	OnlineBanking	901
5	GameConsole	345
6	GameConsole	890
7	GameConsole	901



# Select queries

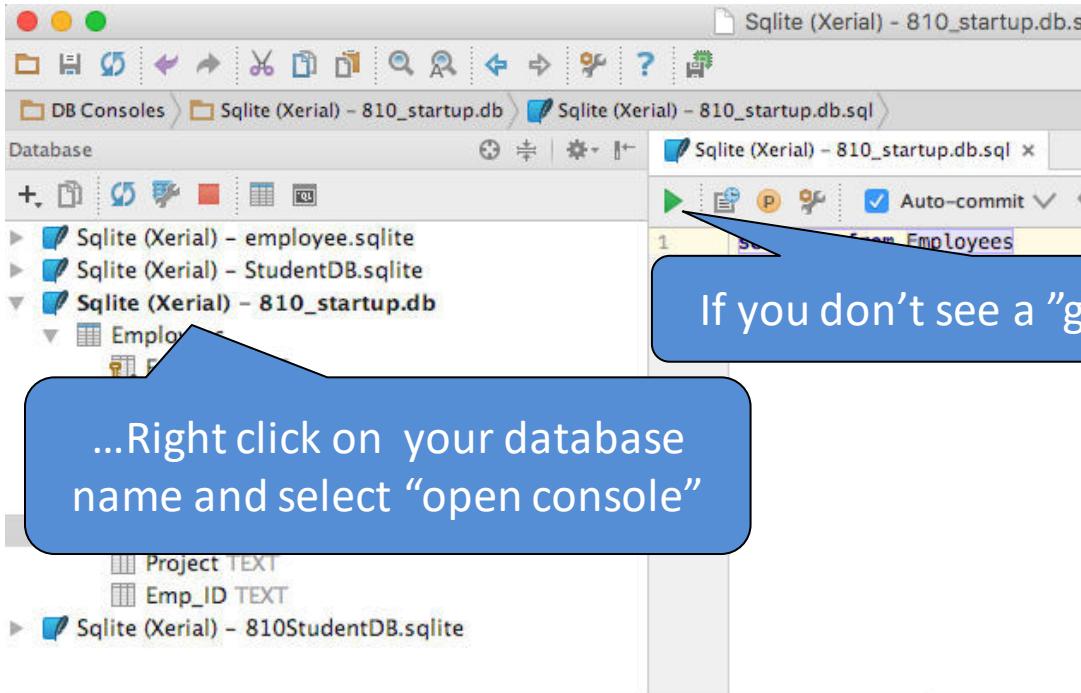
Select chooses a set of rows and columns from one or more tables

**SELECT** *column<sub>1</sub>, ..., column<sub>n</sub>* **FROM** *table*

**select Name, Title from Employees**

	Name	Title
1	Nanda	Developer
2	Sujit	Tester
3	Drashti	Developer
4	Fei	Architect
5	Siddarth	DBA

# Select with DataGrip



The screenshot shows the DataGrip interface. On the left, a database tree displays several databases, including "Sqlite (Xerial) - employee.sqlite", "Sqlite (Xerial) - StudentDB.sqlite", and "Sqlite (Xerial) - 810\_startup.db". A blue callout bubble points to the "Sqlite (Xerial) - 810\_startup.db" node with the text "...Right click on your database name and select “open console”". The main window shows a query editor with the SQL command "SELECT \* FROM Employees" and its results. A green play button icon is visible in the toolbar above the results table. A blue callout bubble points to the results table with the text "If you don't see a “green button”...". Below the results table, another blue callout bubble points to the table header with the text "Query results appear here".

Emp_ID	Name	Title	Salary
1 123	Nanda	Developer	80000
2 234	Sujit	Tester	70000
3 345	Jim	Manager	85000
4 890	Fei	Architect	90000
5 901	Siddarth	DBA	87000



# Select with DataGrip

The screenshot shows the DataGrip interface with a blue callout box containing the text "Write the query, then click the ‘go’ icon". In the SQL editor, the query `select Name, Title from Employees` is highlighted. Another blue callout box contains the text "Highlight your select statement". The results of the query are displayed in a table titled "Result 31" with columns "Name" and "Title". The data is as follows:

	Name	Title
1	Nanda	Developer
2	Sujit	Tester
3	Jim	Manager
4	Fei	Architect
5	Siddarth	DBA

A blue callout box at the bottom right points to the table with the text "Results appear here".

# Special SELECT alternatives

SELECT \* from <Table>

Select all rows from <Table>

SELECT COUNT(\*) FROM <Table>

Select the number of rows in <Table>

SELECT DISTINCT *column<sub>1</sub>*, FROM <Table>

Select all distinct values of column<sub>1</sub> from <Table>

SELECT DISTINCT(Project) FROM Projects

Project	
1	GameConsole
2	OnlineBanking

# Select Aliases

Change the label in the results of a SELECT query

```
SELECT COUNT(*) FROM Employees
```

	"count(*)"	+
1	5	

```
SELECT COUNT(*) AS cnt FROM Employees
```

	cnt	+
1	5	



# Aggregate functions

## Useful aggregate functions for SELECT

Aggregate Function	Returns
AVG(<COLUMN>)	Average of the specified column
COUNT(<COLUMN>)	Number of non-NULL values in the column
COUNT(*)	Number of rows in the table
MAX(<COLUMN>)	Maximum value the specified column
MIN(<COLUMN>)	Minimum value in the specified column
SUM(<COLUMN>)	Sum of the specified column



# Aggregate functions

Employees Table

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

Query	Result
<code>select avg(salary) from Employees</code>	82400
<code>select count&gt;Title) from Employees</code>	5
<code>select count(*) from Employees</code>	5
<code>select max&gt;Title) from Employees</code>	Tester
<code>select min(Name) from Employees</code>	Fei
<code>select sum(Salary) from Employees</code>	412000



# Aggregate functions and GROUP BY

SELECT <COLUMNS> GROUP BY <COLUMNS>

Groups the common values together

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

```
select Title, count(*) as cnt, avg(Salary) as avg_salary  
from Employees group by Title
```

	Title	cnt	avg_salary
1	Architect	1	90000
2	DBA	1	87000
3	Developer	1	80000
4	Manager	1	85000
5	Tester	1	70000

Group the rows by Title  
when computing aggregates



# Select ORDER BY

SELECT <COLUMNS> ORDER BY <COLUMNS> [ASC | DESC]

Orders the output table by the specified columns

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

select Name, Title from Employees order by Salary DESC

	Name	Title
1	Fei	Architect
2	Siddarth	DBA
3	Jim	Manager
4	Nanda	Developer
5	Sujit	Tester

Order the rows in  
descending order by salary



# Select Where

Select a set of rows and columns from one or more tables that meet the specified criteria

**SELECT** *column<sub>1</sub>, ..., column<sub>n</sub>*

**FROM** *table*

**WHERE** *column <operator> <value>*

Operators include =, !=, <>, >, <, >=, <=, !<, !=

select name, Title from Employees

where name = 'Jim'

select \* from Employees

where salary >= 70000



# Data Normalization

Database tables are designed for efficiency

Eliminate redundant data – store it once

- Save space

- Make changes easier because the data is stored only once

Project Example

- Employee: Name, Title, Salary

- Project: Project, Employees assigned to that project



# Option I – (not a great solution)

Represent all data in a single table (“Excel Solution”)

	Name	Title	Salary	Project
1	Nanda	Developer	80000	GameConsole
2	Sujit	Tester	70000	GameConsole
3	Fei	Architect	90000	OnlineBanking
4	Siddarth	DBA	87000	OnlineBanking
5	Jim	Manager	85000	GameConsole
6	Fei	Architect	90000	GameConsole
7	Siddarth	DBA	87000	GameConsole

What happens if Fei is promoted from Architect to Chief Architect and gets a raise?

Must find and update all relevant rows or the data will be inconsistent

Must find and update all relevant records consistently



# Option 2 - Store each item once

Employees

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

Projects

	Project	Emp_ID
1	GameConsole	123
2	GameConsole	234
3	OnlineBanking	890
4	OnlineBanking	901
5	GameConsole	345
6	GameConsole	890
7	GameConsole	901

Any change in Employees  
or Projects requires only  
one change

But how to see the tables  
together???

# Joins

Data is frequently stored in multiple tables

Use keys to join multiple related tables

## ***Primary Key***

One or more columns

Uniquely identifies a row in a table



## ***Foreign Key***

Refers to a Primary Key in a different table

Used to join tables

Not all tables have keys but we can join on any field

# Option 2 - Store each item once

Employees

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

Projects

	Project	Emp_ID
1	GameConsole	123
2	GameConsole	234
3	OnlineBanking	890
4	OnlineBanking	901
5	GameConsole	345
6	GameConsole	890
7	GameConsole	901

Any change in Employees or Projects requires only one change

But how to combine the data across tables???



# Join

Joining tables on primary/foreign keys allows us to combine fields from relevant tables

```
SELECT <columns>
FROM
    <TABLE1> <ALIAS1>
    JOIN <TABLE2> <ALIAS2>
        ON <ALIAS1>.<COLUMN> = <ALIAS2>.<COLUMN>
    JOIN <TABLE3> <ALIAS3>
        ON <ALIAS1>.<COLUMN> = <ALIAS3>.<COLUMN>
```

Join as many  
tables as needed

Join returns a result table with all of the rows meeting the constraints

Join across as many tables as needed

# Option 2 - Store each item once

Employees

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

Primary Key

Projects

	Project	Emp_ID
1	GameConsole	123
2	GameConsole	234
3	OnlineBanking	890
4	OnlineBanking	901
5	GameConsole	345
6	GameConsole	890
7	GameConsole	901

Foreign Key

Primary/Foreign Keys relate records from different tables

# Option 2 - Store each item once

## Employees

	Emp_ID	Name	Title	Salary
1	123	Nanda	Developer	80000
2	234	Sujit	Tester	70000
3	345	Jim	Manager	85000
4	890	Fei	Architect	90000
5	901	Siddarth	DBA	87000

## Projects

	Project	Emp_ID
1	GameConsole	123
2	GameConsole	234
3	OnlineBanking	890
4	OnlineBanking	901
5	GameConsole	345
6	GameConsole	890
7	GameConsole	901

Employees joined  
with Projects

```
select e.Name, e.Title, e.Salary, p.Project
from Employees e
join Projects p on e.Emp_ID=p.Emp_ID
```

	Name	Title	Salary	Project
1	Nanda	Developer	80000	GameConsole
2	Sujit	Tester	70000	GameConsole
3	Fei	Architect	90000	OnlineBanking
4	Siddarth	DBA	87000	OnlineBanking
5	Jim	Manager	85000	GameConsole
6	Fei	Architect	90000	GameConsole
7	Siddarth	DBA	87000	GameConsole



# Answering questions with queries

Who is the highest paid employee?

```
select Name, Title, Salary  
from Employees order by Salary DESC
```

	Name	Title	Salary
1	Fei	Architect	90000
2	Siddarth	DBA	87000
3	Jim	Manager	85000
4	Nanda	Developer	80000
5	Sujit	Tester	70000



How many projects?

```
select count(distinct(Project)) from Projects
```

	"count(distinct(Project))"
1	2





# Answering questions with queries

Who is the highest paid employee?

```
select Name, Title, Salary  
from Employees order by Salary DESC limit 1
```

Limit output to  
1 row

	Name	Title	Salary
1	Fei	Architect	90000



How many projects?

```
select count(distinct(Project)) from Projects
```

	"count(distinct(Project))"
1	2





# Answering questions with queries

What is the total salary for all employees?

```
select sum(Salary) from Employees
```

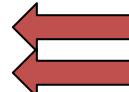
"sum(Salary)"	
1	412000



Which projects have an Architect?

```
select p.Project, e.Title  
from Employees e  
join Projects p on e.Emp_ID=p.Emp_ID  
where e.Title='Architect'
```

	Project	Title
1	OnlineBanking	Architect
2	GameConsole	Architect



# Other types of queries: Join

T1

	Color	Shape
1	Red	Square
2	Blue	Circle
3	Green	Oval

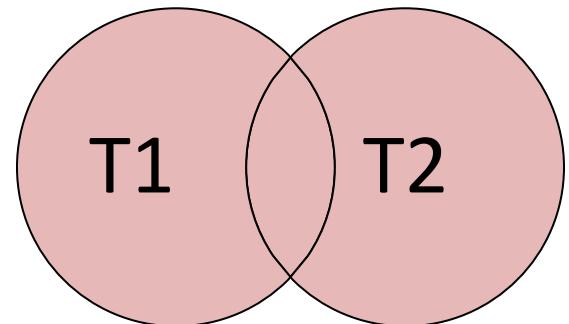
T2

	Color	Size
1	Red	Big
2	Green	Small
3	Yellow	Medium

**select \* from T1 join T2 (No qualifications)**

→ Full cross product of all combinations

	T1.Color	Shape	T2.Color	Size
1	Red	Square	Red	Big
2	Red	Square	Green	Small
3	Red	Square	Yellow	Medium
4	Blue	Circle	Red	Big
5	Blue	Circle	Green	Small
6	Blue	Circle	Yellow	Medium
7	Green	Oval	Red	Big
8	Green	Oval	Green	Small
9	Green	Oval	Yellow	Medium



# Other types of queries: Join on

T1

	Color	Shape
1	Red	Square
2	Blue	Circle
3	Green	Oval

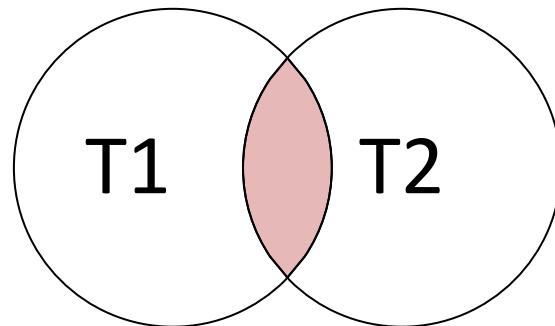
T2

	Color	Size
1	Red	Big
2	Green	Small
3	Yellow	Medium

`select * from T1 join T2 on T1.Color = T2.Color  
 (aka inner join)`

→ Only rows from T1 and T2 where  $T1.Color = T2.Color$

T1.Color	Shape	T2.Color	Size
1	Red	Red	Big
2	Green	Green	Small



# Other types of queries: Left Join

T1

	Color	Shape
1	Red	Square
2	Blue	Circle
3	Green	Oval

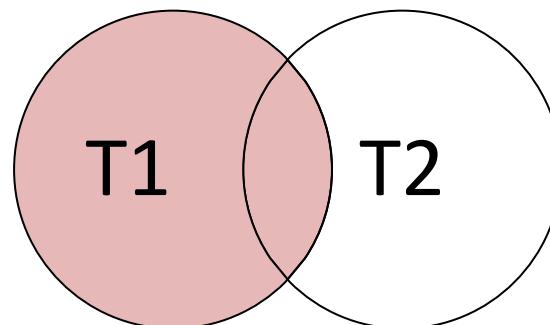
T2

	Color	Size
1	Red	Big
2	Green	Small
3	Yellow	Medium

`select * from T1 left join T2 on T1.Color = T2.Color`

→ All rows from T1 and matching rows from T2 where T1.Color = T2.Color

	T1.Color	Shape	T2.Color	Size
1	Red	Square	Red	Big
2	Blue	Circle	<null>	<null>
3	Green	Oval	Green	Small



# Accessing Databases with Python

Python provides APIs to interact with databases

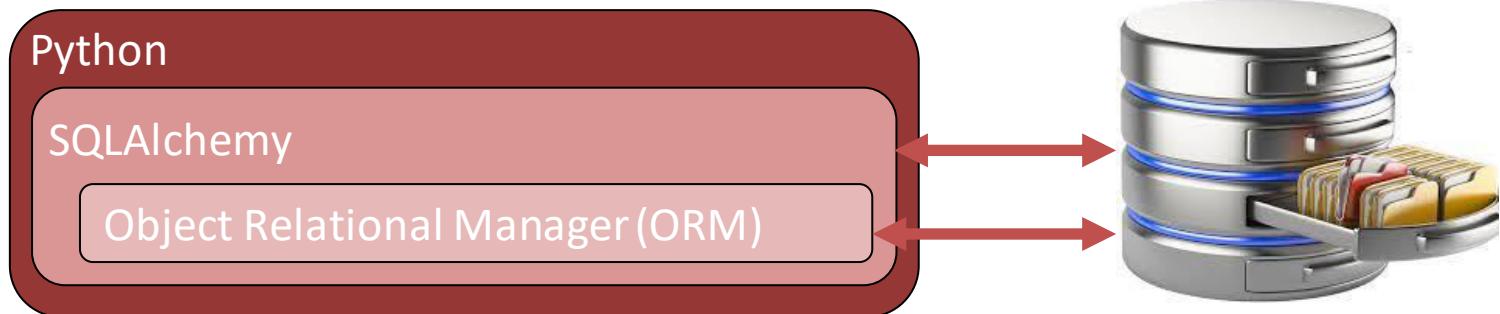
Approaches:

Database specific APIs using module for each database

SQLAlchemy

Add a logical layer between Python code and the database

Object Relational Manager (ORM)

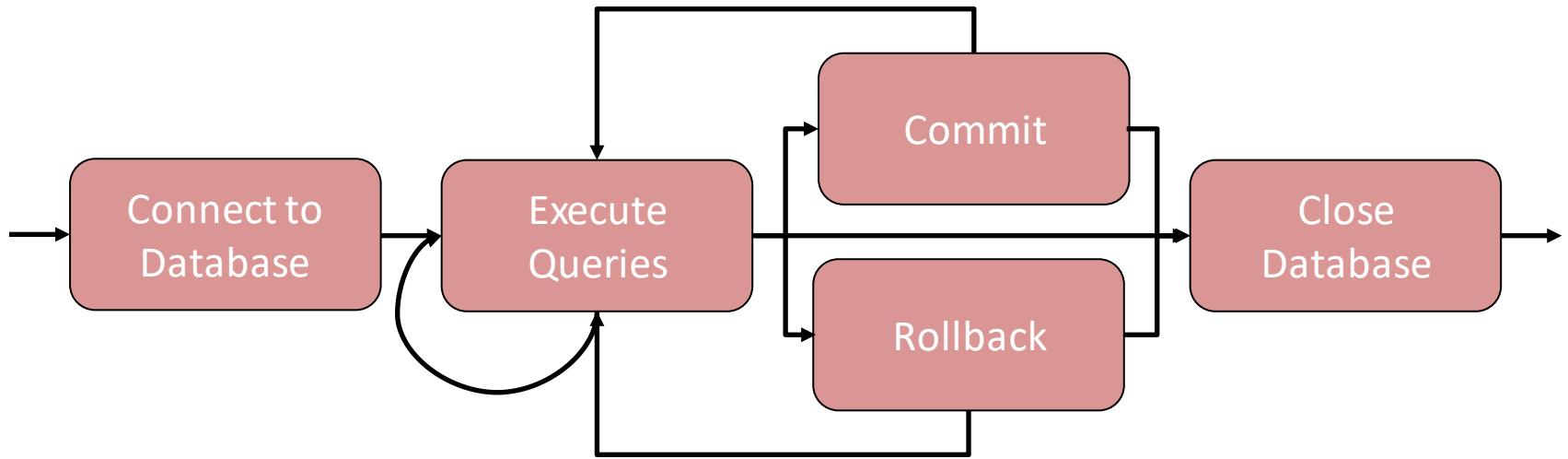




# Python Database APIs

Action	Semantics
Connect	Open a connection to the database
Create	Create tables, insert rows
Read	Select rows
Update	Update existing table rows
Delete	Delete existing table rows
Commit	Commit transaction changes
Roll back	Undo changes since last commit
Close	Close the database connection

# Database Operations and Python APIs





# Start-up company sample database

Employees Table

Emp_ID	Name	Title	Salary
123	Nanda	Developer	80000
234	Sujit	Tester	70000
345	Jim	Manager	85000
890	Fei	Architect	90000
901	Siddarth	DBA	87000

Projects Table

Project	Emp_ID
GameConsole	123
GameConsole	234
OnlineBanking	890
OnlineBanking	901
GameConsole	345
GameConsole	890
GameConsole	901



# Connect

SQLite supports both memory and file-based databases

```
import sqlite3

DB_FILE: str = "/Users/jrr/Documents/Stevens/810/examples/810_startup.db"

db: sqlite3.Connection = sqlite3.connect(DB_FILE)
```

*connect(path)* opens the database at path or in memory if *path == ':memory:'*

***Beware:*** SQLite creates a new database file if you specify a non-existent file



# Select queries

```
for row in db.execute("select Emp_id, Name from Employees"):  
    print(row)
```

```
('123', 'Nanda')  
('345', 'Jim')  
('890', 'Fei')  
('901', 'Siddarth')  
('234', 'Sujit')
```

Execute arbitrary queries

*Execute(query) returns a sequence of tuples with the rows from the query*



# Select queries

```
query: str = """select Project, count(*) as cnt
              from Projects
              group by Project"""
for row in db.execute(query):
    print(row)
```

```
('GameConsole', 5)
('OnlineBanking', 2)
```

*Execute any valid query,  
including aggregations*



# Select queries

```
query: str = """select p.Project, avg(e.Salary) as avg_salary  
            from Projects p join Employees e on p.Emp_ID=e.Emp_ID  
            group by p.Project"""  
  
for row in db.execute(query):  
    print(row)  
  
('GameConsole', 82400.0)  
('OnlineBanking', 88500.0)
```

*Execute any valid query,  
including joins, group by,  
etc.*

*Write and test queries in  
DataGrip then copy the  
SQL to Python*



# Parameterized queries

## How to add a parameter to a query?

```
who: str = input("Enter employee name: ")
query: str = f"select Name, Salary from Employees where Name='{who}'"
for row in db.execute(query):
    print(row)
```

```
Enter employee name: Nanda
('Nanda', 80000)
```

*The query is just a string, so create a string to represent the query*

## What could possibly go wrong?

*Simple, but wrong... and dangerous...*



# SQL Injection Attacks

## Scenario:

- A bad actor is authorized to see only his own salary but wants to know all salaries
- A naïve developer concatenates user input as part of the query
- The bad actor injects extra SQL in the input to change the query

```
user_input: str="Nanda' or 'gotcha'='gotcha"  
query: str = f"select Name, Salary from Employees where Name='{user_input}'"  
print(query)
```

```
select Name, Salary from Employees where Name='Nanda' or 'gotcha'='gotcha'
```

Matches ALL rows



# SQL Injection Attack on Parameterized Query

```
who: str = input("Enter employee name: ")
query: str = f"select Name, Salary from Employees where Name='{who}'"
print(f"Running query: {query}")
for row in db.execute(query):
    print(row)
```

Enter employee name: Nanda' or 'gotcha'='gotcha

Running query: select Name, Salary from Employees where Name='Nanda' or 'gotcha'='gotcha'  
('Nanda', 80000)  
('Jim', 85000)  
('Fei', 90000)  
('Siddarth', 87000)  
('Sujit', 70000)

*Looks okay, right???*

*Bad actor changed  
the query from input*

*Results from ALL employees*

*WAIT!!!  
That's not what we intended!*



# Preventing SQL Injection Attack on Parameterized Query

Include ‘?’ in the query for each parameter and pass the parameters separately as a tuple

execute checks the parameters to prevent SQL Injection attacks

```
who: str = input("Enter employee name: ")
query: str = f"select Name, Salary from Employees where Name=?"
args: Tuple[str] = (who,)
for row in db.execute(query, args):
    print(row)
```

Enter employee name: Nanda  
('Nanda', 80000)

*Specify ‘?’ in the query for each parameter*

*Pass the arguments to execute() as a tuple*



# Insert queries

## Insert rows into tables using computed values

```
query: str = """insert into Employees (Emp_id, Name, Title, Salary)
              values (?, ?, ?, ?)
            """
args: Tuple[str, str, str, int] = ('678', 'Maha', 'Developer', 85000)
db.execute(query, args)
db.commit()

for row in db.execute("select * from Employees"):
    print(row)

('123', 'Nanda', 'Developer', 80000)
('345', 'Jim', 'Manager', 85000)
('890', 'Fei', 'Architect', 100000)
('901', 'Siddarth', 'DBA', 87000)
('234', 'Sujit', 'Tester', 70000)
('678', 'Maha', 'Developer', 85000)
```

*The new row won't appear in queries until commit()*

*The new row*



# Update queries

```
query: str = """select Emp_id, Salary from Employees where Name='Fei'"""
for row in db.execute(query):
    print(row)
('890', 90000)
```

*Fei earned a raise*

```
query: str = """Update Employees set Salary=100000 where Emp_id=890"""
db.execute(query)
db.commit()
```

*The new row won't appear  
in queries until commit()*

```
query: str = """select Emp_id, Salary from Employees where Name='Fei'"""
for row in db.execute(query):
    print(row)
('890', 100000)
```

*The updated row*



# Delete queries

```
query = "delete from Employees where Emp_id=?"
args = ('234',)
db.execute(query, args)
db.commit()
```

```
for row in db.execute("select * from Employees"):
    print(row)
```

```
('123', 'Nanda', 'Developer', 80000)
('345', 'Jim', 'Manager', 85000)
('890', 'Fei', 'Architect', 100000)
('901', 'Siddarth', 'DBA', 87000)
('678', 'Maha', 'Developer', 85000)
```

*Emp\_ID 234 has  
been deleted*



# Close

Use `close()` to close the connection to the database

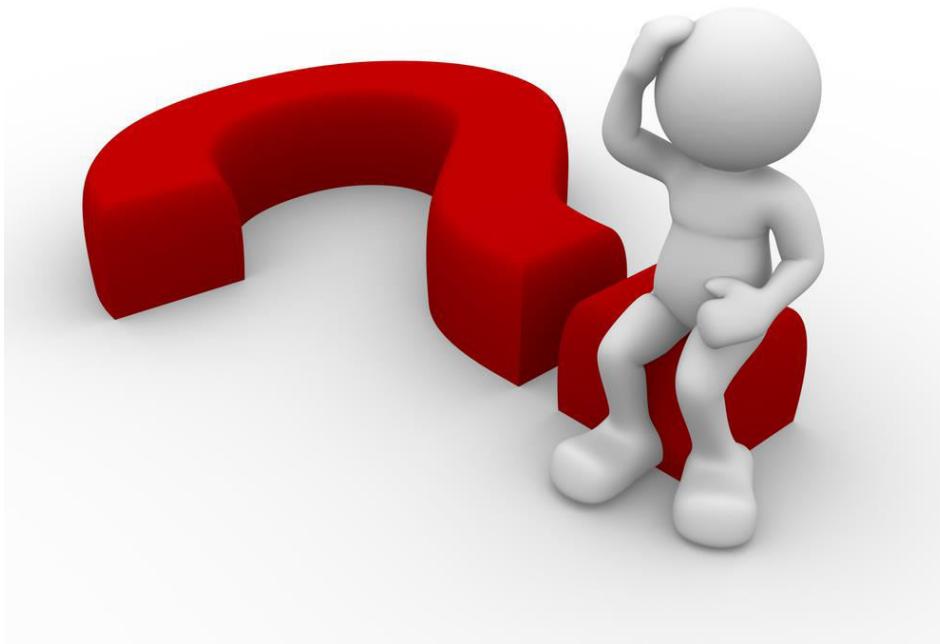
```
db.close()
```



# SQLite Python API Summary

Action	Semantics
Connect	Open a connection to the database
Create	Create tables, insert rows
Read	Select rows
Update	Update existing table rows
Delete	Delete existing table rows
Commit	Commit transaction changes
Roll back	Undo changes since last commit
Close	Close the database connection

# Questions?





# SSW-810: Software Engineering Tools and Techniques

*Building Web Sites with Python  
and Flask*

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from:

- <http://www.tutorialspoint.com/dbms/>
- <https://www.tutorialspoint.com/html/>
- <https://gertjanvanhethofblog.wordpress.com/2016/02/10/exploring-flask-on-raspberry-pi/>
- <http://flask.pocoo.org/>



# Today's topics

Tying it all together

HTML

Integrating web pages and Python

Flask

Building a simple web site to display student information

Populating forms with Python and Flask





# What problem are we trying to solve?

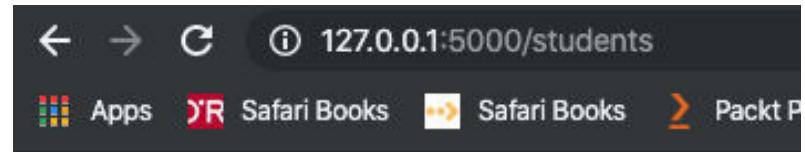
**Goal:** build a simple website

Display data from our SQLite database

## Limitations:

**Not** a lesson in HTML

**Not** a lesson in good web design

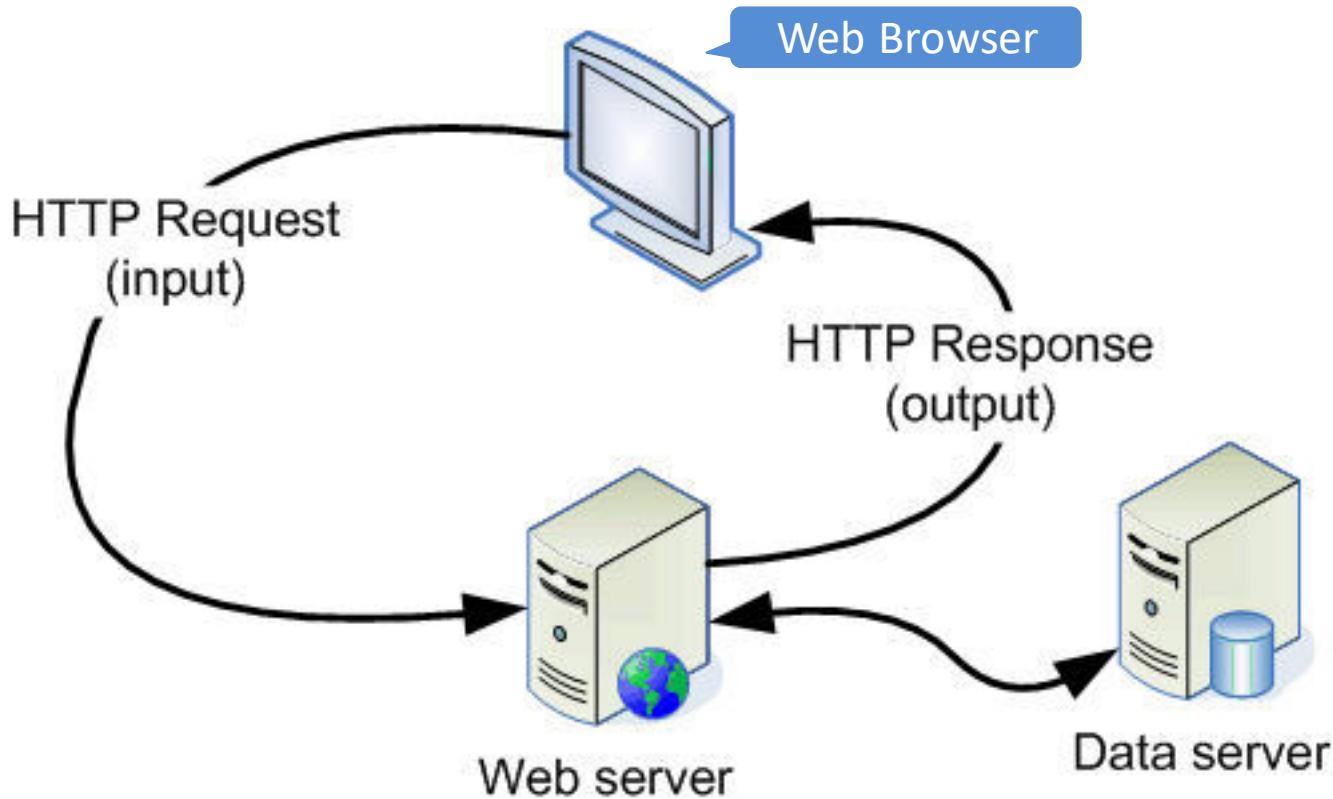


## Stevens Repository

Number of completed courses by Student

CWID	Name	Major	Completed Courses
10103	Jobs, S	SFEN	2
10115	Bezos, J	SFEN	2
10183	Musk, E	SFEN	2
11714	Gates, B	CS	3

# Oversimplified Web Overview



<http://testpad.asna.com/tutorials/Images/DataServerWebServer.gif>

# Static and Dynamic HTML

Many HTML pages have static content

hello.html

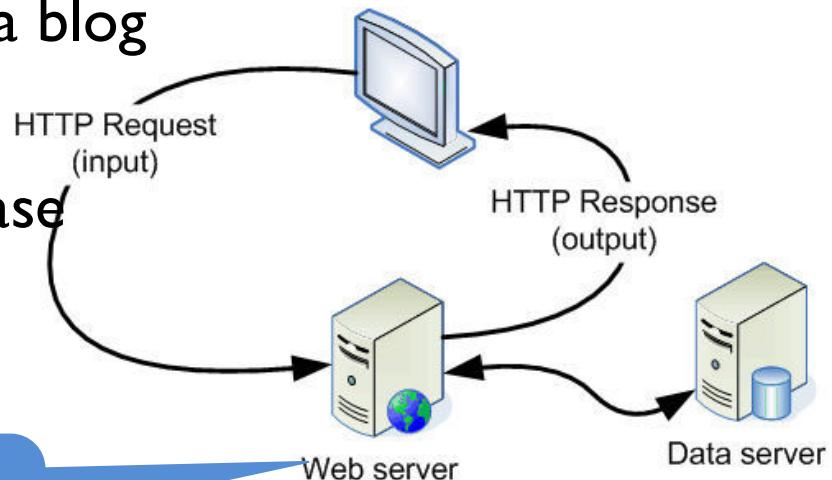
Kitten videos

Other pages generate HTML dynamically

Display the latest messages from a blog

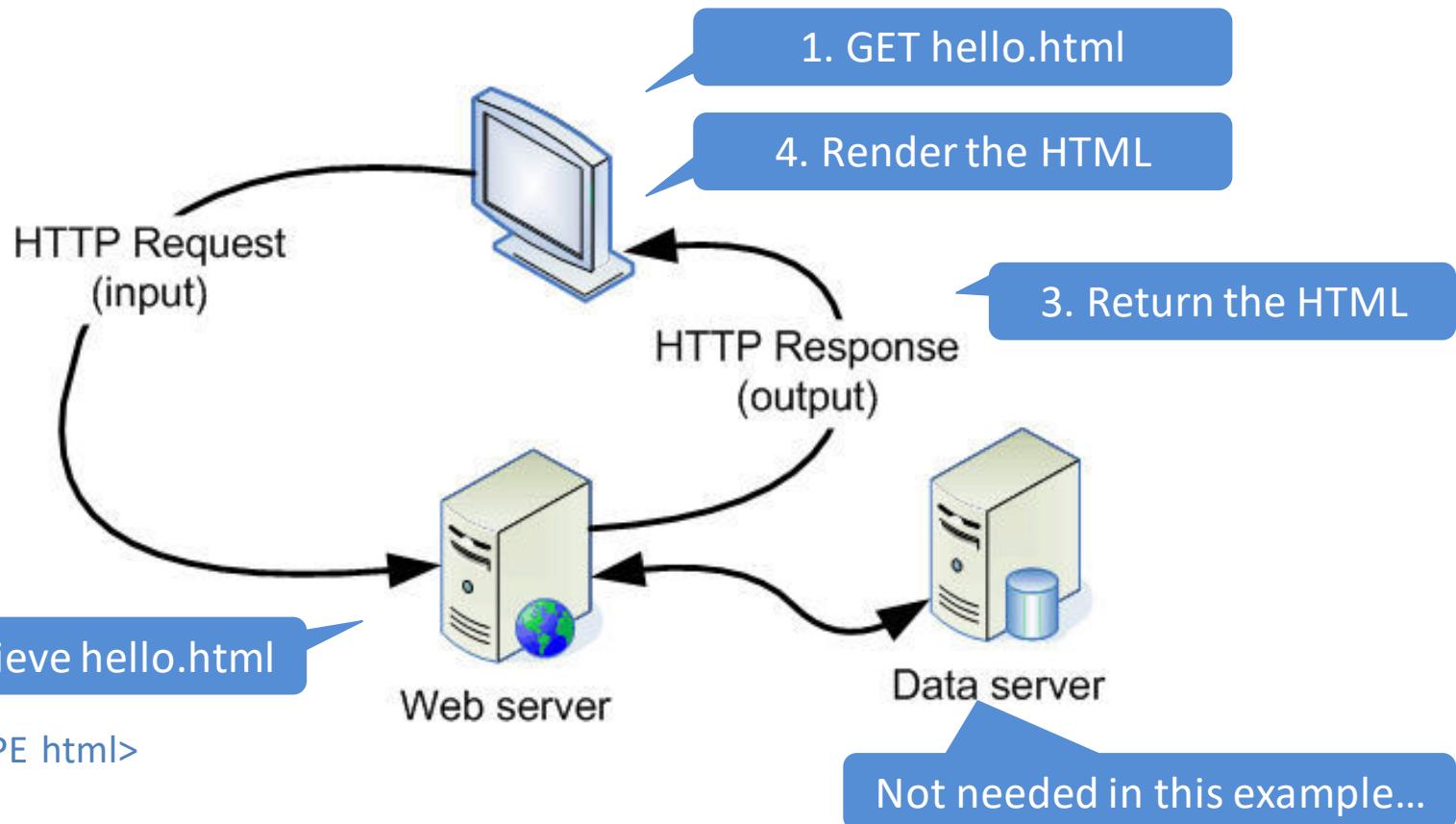
Display the latest Tweets

Display information from a database



The Web Server may generate the HTML dynamically using information retrieved from a database or other source

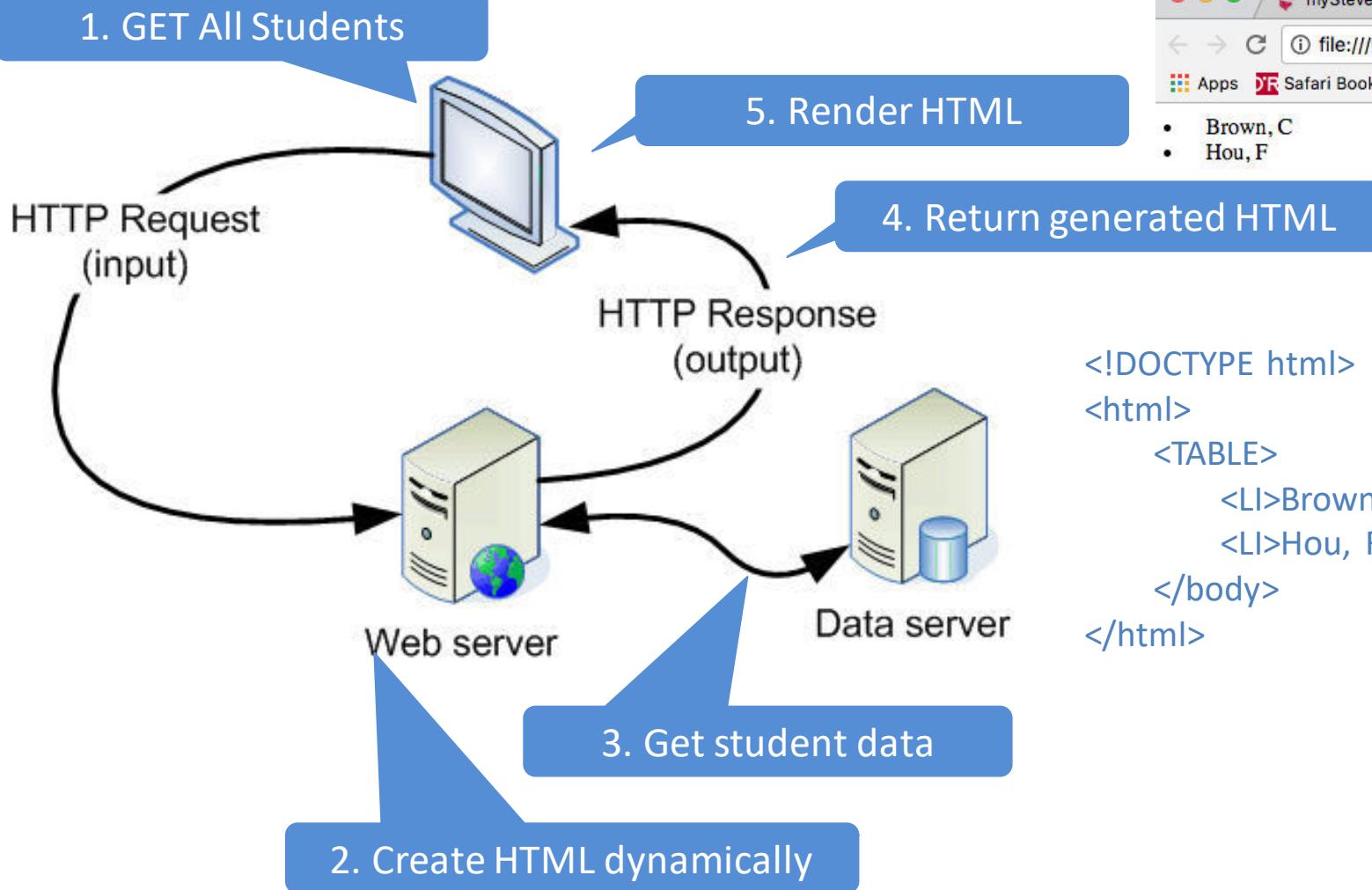
# Oversimplified Web Overview



```
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

<http://testpad.asna.com/tutorials/Images/DataServerWebServer.gif>

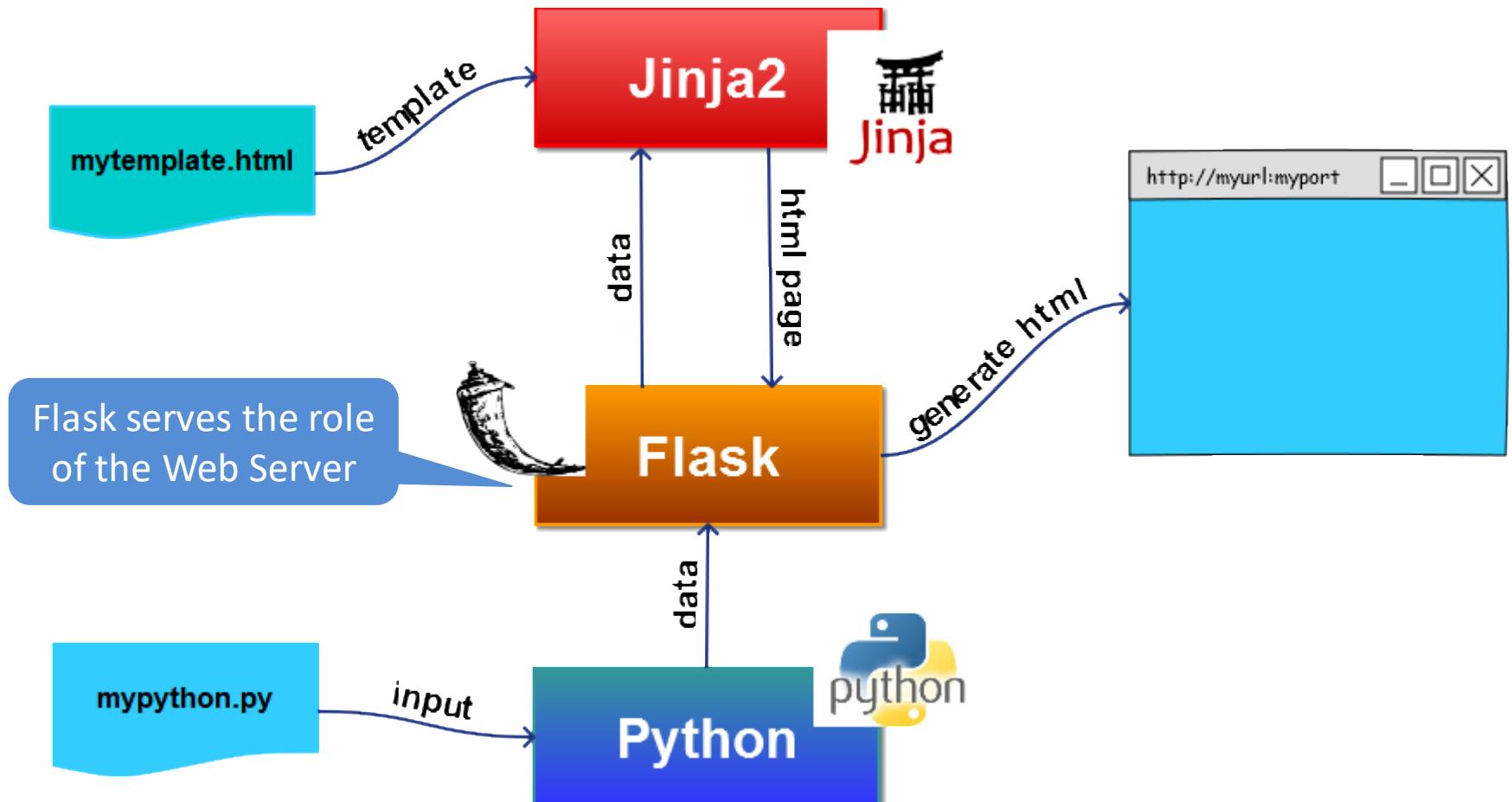
# Dynamic Web Pages



```
<!DOCTYPE html>
<html>
  <TABLE>
    <LI>Brown, C</LI>
    <LI>Hou, F</LI>
  </body>
</html>
```

<http://testpad.asna.com/tutorials/Images/DataSourceWebServer.gif>

# Python, Flask, Jinja2



<https://gertjanvanhethofblog.wordpress.com/2016/02/10/exploring-flask-on-raspberry-pi/>



# Hello World with Flask

Dish: \_\_\_\_\_

## Recipe

Serves: \_\_\_\_\_

1. Install Flask and its supporting packages
2. Create a web server program with Python and Flask  
Return “Hello world” to the browser
3. Run our web server program
4. Point to our web server with a browser
5. Say “Hello”!



# Installing the packages

```
$ pip3 install flask
```

Mac/Unix

```
C:\> pip3 install flask
```

Windows

Add the Flask package to your Python environment



# Background: Python decorators

Python **decorators** provide additional information about a function or class definition

Decorators augment the definition of a function or class without changing the definition of the function or class

```
@staticmethod
```

Specify that `foo` is a static method

```
def foo(): """ This is a static method """
```

```
@app.route('/hello')
def hello_page() -> str:
    return "Hello world!"
```

Call `hello_page()` function when the user requests the 'hello' page. Return HTML to be rendered in the web browser

Return any valid HTML to be rendered by the browser



# Step 1: Flask Hello World

Create hello.py which will be our web server

✚ hello.py > ...

```
1  """ Testing Flask installation """
2  from flask import Flask, render_template
3
4  app: Flask = Flask(__name__)
5
6  @app.route('/Hello')
7  def hello() -> str:
8      |     return "Hello world! This is Flask!"
9
10 app.run(debug=True)
```

Call 'hello()' when the browser requests the /Hello page

The return value is HTML that will be rendered by the browser

Automatically restart Flask if the file changes



# Step 2: Running the Flask Web Server

hello.py > ...

```
1  """ Testing Flask installation """
2  from flask import Flask, render_template
3
4  app: Flask = Flask(__name__)
5
6  @app.route('/Hello')
7  def hello() -> str:
8      return "Hello world! This is Flask!"
9
10 app.run(debug=True)
```

hello.py program from previous slide

Run your hello.py program from a shell/command window, or an IDE

```
(py38) jrr@MacBook-Pro-3 flask % python hello.py
* Serving Flask app "hello" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with fsevents reloader
* Debugger is active!
* Debugger PIN: 194-123-608
127.0.0.1 - - [11/Apr/2020 08:58:50] "GET /Hello HTTP/1.1" 200 -
```

Hit ^C to stop the process to shut down the server after testing or to make changes to the program

Someone accessed your Hello page with a GET request

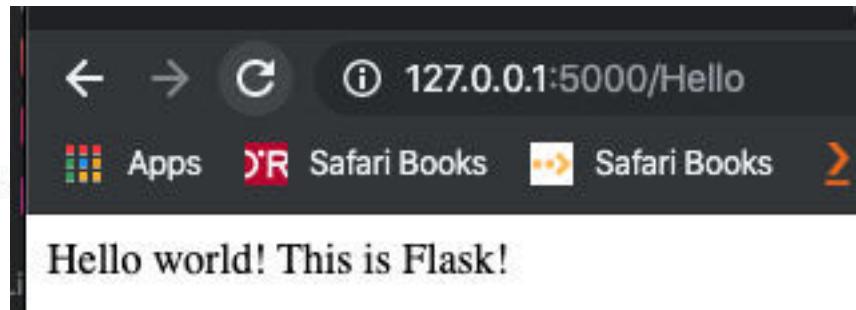


# Step 3: Accessing the Flask Web Server

hello.py > ...

```
1  """ Testing Flask installation """
2  from flask import Flask, render_template
3
4  app: Flask = Flask(__name__)
5
6
7
8
9
10 app.run(debug=True)
```

2. Open your browser and enter  
127.0.0.1:5000/Hello as the URL



```
(py38) jrr@MacBook-Pro-3 flask % python hello.py
 * Serving Flask app "hello" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/Hello (D)
 * Restarting with fsevents ...
 * Debugger is active!
 * Debugger PIN: 194-123-608
127.0.0.1 - - [11/Apr/2020 08:58:50] "GET /Hello HTTP/1.1" 200 -
```

1. Run your hello.py program from  
a shell/command window of IDE

3. Hit ^C to stop the  
Python program

Recreate this scenario on your laptop to verify that  
Flask is installed and working properly.



# Add Goodbye page to our site

Our Flask web server can support many pages

hello.py > ...

```
1  """ Testing Flask installation """
2  from flask import Flask, render_template
3
4  app: Flask = Flask(__name__)
5
6  @app.route('/Hello')
7  def hello() -> str:
8      |     return "Hello world! This is Flask!"
9
10 @app.route('/Goodbye')
11 def see_ya() -> str:
12     |     return "See you later!"
13
14 app.run(debug=True)
```

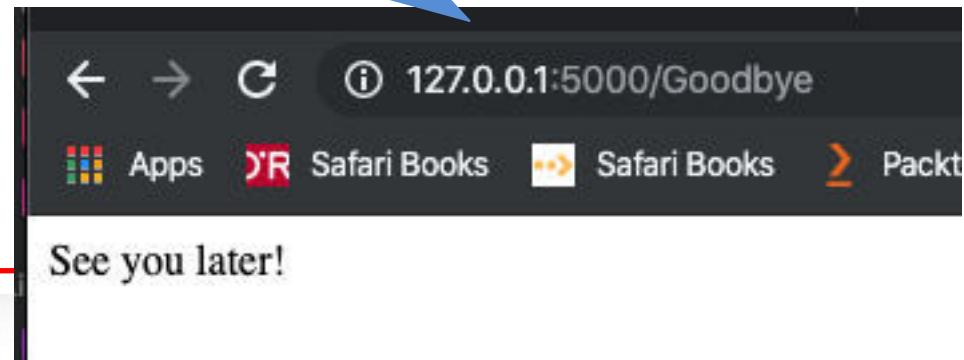
Each @app.route(URL)  
displays a different web page

# Add Goodbye page to our site

hello.py > ...

```
1  """ Testing Flask installation """
2  from flask import Flask, render_template
3
4  app: Flask = Flask(__name__)
5
6  @app.route('/Hello')
7  def hello() -> str:
8      return "Hello world! This is Flask."
9
10 @app.route('/Goodbye')
11 def see_ya() -> str:
12     return "See you later!"
13
14 app.run(debug=True)
```

Test the new feature!



```
(py38) jrr@MacBook-Pro-3 flask % python hello.py
```

```
* Serving Flask app "hello" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with fsevents reloader
* Debugger is active!
* Debugger PIN: 194-123-608
```

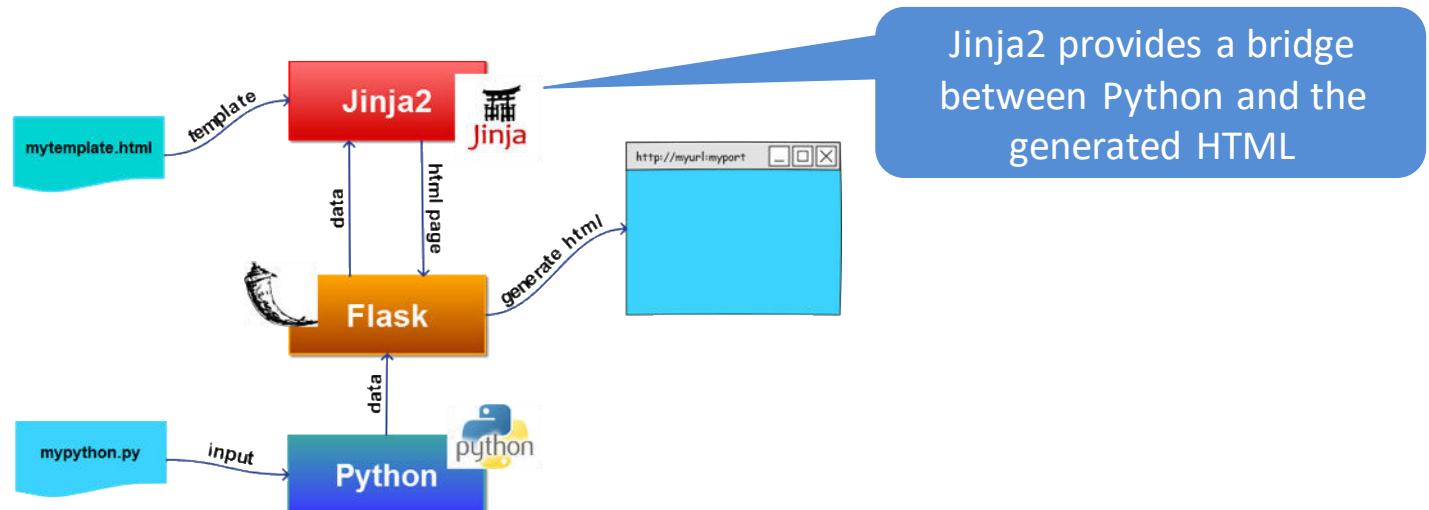
```
127.0.0.1 - - [11/Apr/2020 09:12:27] "GET /Goodbye HTTP/1.1" 200 -
```

# Making pages more interesting

We want to support more interesting pages

e.g. display variables from Python in the HTML page  
(the first step in dynamic web content)

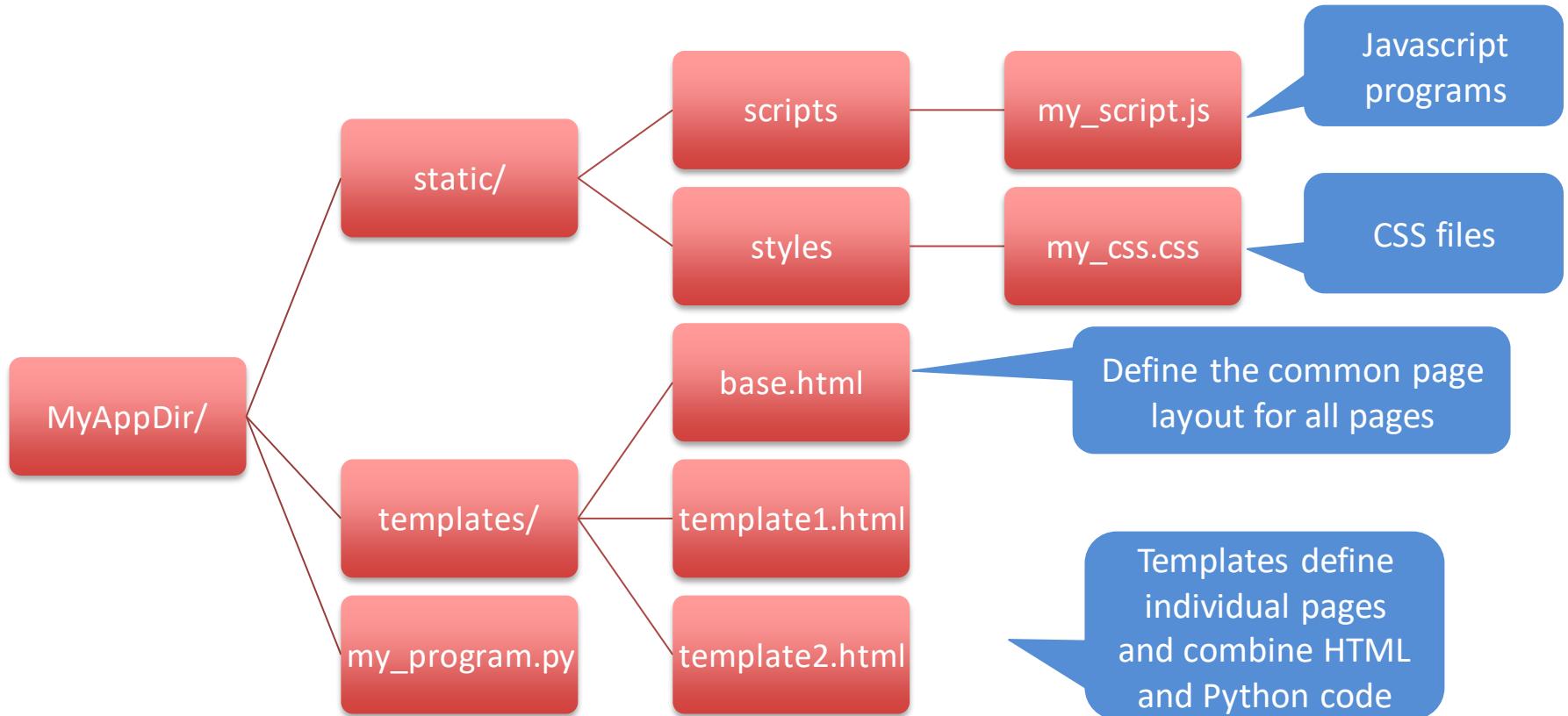
Flask works with many components, including Jinja2



<https://gertjanvanhethofblog.wordpress.com/2016/02/10/exploring-flask-on-raspberry-pi/>

# Flask application directory structure

Flask/Jinja2 **requires** a specific directory structure





# HTML Intro

## HTML5 – HyperText Markup Language

A language to specify the content and the appearance of pages

templates > `hello.html`

```
1  <!DOCTYPE html>
2  <head>
3  |   <title>Hello World!</title>
4  |</head>
5  <body>
6  |<H1>Hello World!</H1>
7
8  This is a very simple HTML document
9  |</body>
10 |</html>
```

`<tag>` provides layout hints `</tag>`

**Hello World!**

This is a very simple HTML document



# Jinja2 base.html

templates/base.html defines a base template that will be inherited by other pages in your project

This is especially helpful when using CSS or creating multiple pages with a consistent look and feel

templates > base.html

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>{{ title }}</title>
5      </head>
6      <body>
7          {% block body %}
8          {% endblock %}
9      </body>
10 
```

{{ title }} will be replaced by a value provided by your program in the HTML

Everything between  
 { % block body % } and  
 { % endblock % }  
 will be replaced by a template page  
 that inherits from this page



# Jinja2 templates

templates/your\_template.html inherits from the base template in base.html and adds HTML to your specific page

templates > parameters.html

```
1  {% extends 'base.html' %}  
2  
3  {% block body %} → Define the block body that will plug  
4  
5  <h2>{{ my_header }}</h2>  
6  My Python parameter in HTML: '{{ my_param }}'  
7  {% endblock %}
```

{% statement %} is interpreted by Jinja2

→ {{ my\_header}} and {{ my\_param }} will be replaced by values from your Python program



# Jinja2 templates

## Apply the template from your Python program

hello.py > ...

```
1  """
2      Testing Flask installation
3  """
4  from flask import Flask, render_template
5
6 app: Flask = Flask(__name__)
7
8 @app.route('/Hello')
9 def hello() -> str:
10     return "Hello world! This is Flask!"
11
12 @app.route('/Goodbye')
13 def see_ya() -> str:
14     return "See you later!
15
16 @app.route('/sample_template')
17 def template_demo() -> str:
18     return render_template('parameters.html',
19                           my_header="My Stevens Repository",
20                           my_param="My custom parameter")
21
22 app.run(debug=True)
```

Import render\_template from flask

Add a new web page

render\_template(template, params) combines the specified template.html, base.html, and parameter values from your Python program to create an HTML document

# Jinja2 templates

templates > base.html

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>{{ title }}</title>
5      </head>
6      <body>
7          {% block body %}
8          {% endblock %}
9      </body>
10 
```

1. templates/base.html

```

hello.py > ...
1  """
2      Testing Flask installation
3  """
4  from flask import Flask, render_template
5
6  app: Flask = Flask(__name__)
7
8  @app.route('/Hello')
9  def hello() -> str:
10     return "Hello world! This is Flask!"
11
12 @app.route('/Goodbye')
13 def see_ya() -> str:
14     return "See you later!"
15
16 @app.route('/sample_template')
17 def template_demo() -> str:
18     return render_template('parameters.html',
19                           my_header="My Stevens Repository",
20                           my_param="My custom parameter")
21
22 app.run(debug=True)

```

3. hello.py

Passed to the template

templates > parameters.html

```

1  {% extends 'base.html' %} 2
3  {% block body %} 4
5  <h2>{{ my_header }}</h2> 6  My Python parameter in HTML: '{{ my_param }}' 7  {% endblock %}

```

2. templates/parameters.html

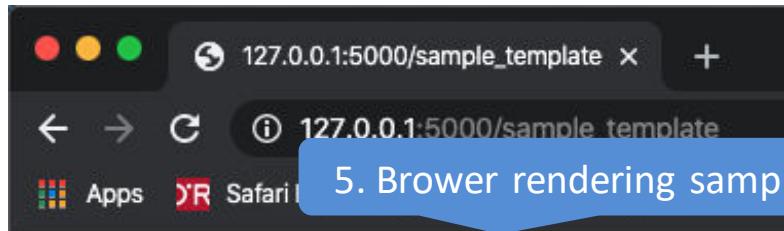
(py38) jrrr@MacBook-Pro-3 flask % python hello.py

```

* Serving Flask app "hello" (lazy loading)
* Environment: production
WARNING: This is a development server
Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with fsevents reloader
* Debugger is active!
* Debugger PIN: 194-123-608

```

4. Terminal running hello.py



5. Brower rendering sample\_template

## My Stevens Repository

My Python parameter in HTML: 'My custom parameter'



# Moving Student Data to the Web

Goal: present student information from student repository as a web page

## Stevens Repository

Number of completed courses by Student

CWID	Name	Major	Completed Courses
10115	Bezos, J	SFEN	2
11714	Gates, B	CS	3
10103	Jobs, S	SFEN	2
10183	Musk, E	SFEN	2

We need to generate this  
web page dynamically  
from our Python program



# Intro to HTML Tables

HTML for four column, two row table

templates > sample\_table.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <body>
4  <table border="1">
5      <th>CWID</th>
6      <th>Name</th>
7      <th>Major</th>
8      <th>Completed Courses</th>
9      <tr>
10         <td>cwid 1</td>
11         <td>name 1</td>
12         <td>major 1</td>
13         <td>complete 1</td>
14     </tr>
15     <tr>
16         <td>cwid 2</td>
17         <td>name 2</td>
18         <td>major 2</td>
19         <td>complete 2</td>
20     </tr>
21 </table>
22 </body>
23 </html>
```

TH: Table Header (column)

TR: Table Row

TD: Table Data  
(1 column in 1 row)

Table rendered in the browser

CWID	Name	Major	Completed Courses
cwid 1	name 1	major 1	complete 1
cwid 2	name 2	major 2	complete 2

Goal: update our template to generate  
HTML for an arbitrary list of student data



# Displaying tables with Flask & Jinja2

We've seen HTML for tables

We know how to pass parameters from Python to Jinja2

How to display Python containers in HTML?

Store values in a dict and pass to Jinja2 as parameters

Jinja2 supports iterating through sequences

```
<table>
    {%
        for item in sequence %
    <tr>
        <td> {{ item }} <\td>
    <\tr>
    {%
        endfor %
    </table>
```

Jinja2 iterates through the items in a sequence passed from Python

Use the value from the sequence

End of the Jinja2 for loop



# Displaying tables with Flask & Jinja2

## Update templates/student\_courses.html

templates > student\_courses.html

```
1  {% extends 'base.html' %}  
2  {% block body %}  
3  <h2>{{ title }}</h2>  
4  <h3> {{ table_title }}</h3>  
5  <table border="1">  
6      <th>CWID</th>          Surround the table with borders  
7      <th>Name</th>  
8      <th>Major</th>  
9      <th>Completed Courses</th>  
10     {% for student in students %}  We don't know, and don't care, how  
11         <tr>                         many rows in students  
12             <td>{{ student.cwid }}</td>  <tr> Table Row </tr>  
13             <td>{{ student.name }}</td>  
14             <td>{{ student.major }}</td>  
15             <td>{{ student.complete }}</td>  
16         </tr>  
17     {% endfor %}                    Add a row to the table with 4  
18   </table>                        columns for each student in the list  
19   {% endblock %}                      of dicts passed in from Python
```



# Displaying tables with Flask & Jinja2

## Update static\_table.py

static\_table.py > ...

```
1  """
2  | Example of a static table from Python with Flask
3  """
4  from flask import Flask, render_template
5  from typing import Dict, List
6
7  app: Flask = Flask(__name__)
8
9  @app.route('/students')
10 def students_summary() -> str:
11     data: List[Dict[str, str]] = [ # partial list of students
12         {'cwid': '10115',
13          'name': 'Bezos, J',
14          'major': 'SFEN',
15          'complete': '2',
16          },
17         {'cwid': '10102',
18          'name': 'Gates, B',
19          'major': 'CS',
20          'complete': '3',
21         },
22     ]
23
24     return render_template('student_courses.html',
25                           title='Stevens Repository',
26                           table_title='Student Summary',
27                           students=data)
28
29 app.run(debug=True)
```

Jinja2 supports JSON format for complex data

Pass in a list of students in JSON format

Python represents JSON as a dictionary with keys and values

Access each key/value separately in template/\*.html

Pass the list of dicts to the HTML template



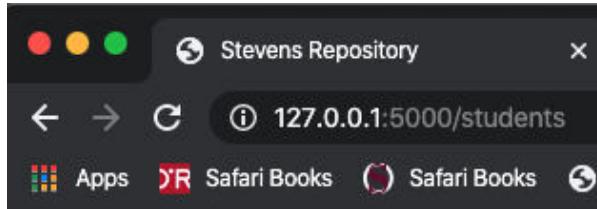
# Displaying tables with Flask & Jinja2

templates > student\_courses.html

```
1  {% extends 'base.html' %} 
2  {% block body %} 
3  <h2>{{ title }}</h2> 
4  <h3> {{ table_title }}</h3> 
5  <table border="1"> 
6      <th>CWID</th> 
7      <th>Name</th> 
8      <th>Major</th> 
9      <th>Completed Courses</th> 
10     {% for student in students %} 
11         <tr> 
12             <td>{{ student.cwid }}</td> 
13             <td>{{ student.name }}</td> 
14             <td>{{ student.major }}</td> 
15             <td>{{ student.complete }}</td> 
16         </tr> 
17     {% endfor %} 
18 </table> 
19 {% endblock %}
```

templates/student\_courses.html

Browser Window



## Stevens Repository

### Student Summary

CWID	Name	Major	Completed Courses
10115	Bezos, J	SFEN	2
10102	Gates, B	CS	3

static\_table.py

```
9  @app.route('/students') 
10 def students_summary() -> str: 
11     data: List[Dict[str, str]] = [ # partial list of students 
12         {'cwid': '10115', 
13          'name': 'Bezos, J', 
14          'major': 'SFEN', 
15          'complete': '2', 
16        }, 
17        {'cwid': '10102', 
18          'name': 'Gates, B', 
19          'major': 'CS', 
20          'complete': '3', 
21        }, 
22    ] 
23 
24    return render_template('student_courses.html', 
25                           title='Stevens Repository', 
26                           table_title='Student Summary', 
27                           students=data) 
28 
29 app.run(debug=True)
```



# When things go wrong... missing rows

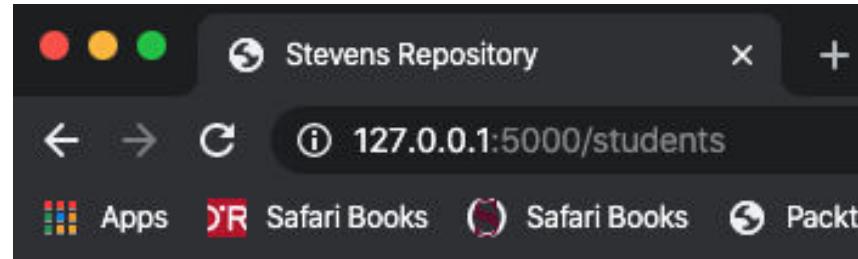
templates > student\_courses.html

```
1  {% extends 'base.html' %} 
2  {% block body %} 
3  <h2>{{ title }}</h2> 
4  <h3> {{ table_title }}</h3> 
5  <table border="1"> 
6      <th>CWID</th> 
7      <th>Name</th> 
8      <th>Major</th> 
9      <th>Completed Courses</th> 
10     {% for student in students %} 
11         <tr> 
12             <td>{{ student.cwid }}</td> 
13             <td>{{ student.name }}</td> 
14             <td>{{ student.major }}</td> 
15             <td>{{ student.complete }}</td> 
16         </tr> 
17     {% endfor %} 
18 </table> 
19 {% endblock %}
```

buggy\_static\_table.py > ...

```
9  @app.route('/students') 
10 def students_summary() -> str: 
11     data: List[Dict[str, str]] = [ # partial list of students 
12         {'cwid': '10115', 
13          'name': 'Bezos, J', 
14          'major': 'SFEN', 
15          'complete': '2', 
16        }, 
17        {'cwid': '10102', 
18          'name': 'Gates, B', 
19          'major': 'CS', 
20          'complete': '3', 
21        }, 
22    ] 
23 
24    return render_template('student_courses.html', 
25                           title='Stevens Repository', 
26                           table_title='Student Summary', 
27                           oops=data) 
28 
29 app.run(debug=True)
```

templates/student\_courses.html  
unchanged



## Stevens Repository

### Student Summary

Missing data rows!

CWID	Name	Major	Completed Courses
10115	Bezos, J	SFEN	2
10102	Gates, B	CS	3

Oops! The variable names must match  
to pass data from .py to .html

# When things go wrong... missing columns

templates > student\_courses.html

```

1  {% extends 'base.html' %}
2  {% block body %}
3  <h2>{{ title }}</h2>
4  <h3> {{ table_title }}</h3>
5  <table border="1">
6      <th>CWID</th>
7      <th>Name</th>
8      <th>Major</th>
9      <th>Completed Courses</th>
10     {% for student in students %}
11         <tr>
12             <td>{{ student.cwid }}</td>
13             <td>{{ student.name }}</td>
14             <td>{{ student.major }}</td>
15             <td>{{ student.complete }}</td>
16         </tr>
17     {% endfor %}
18 </table>
19 {% endblock %}
```

templates/student\_courses.html

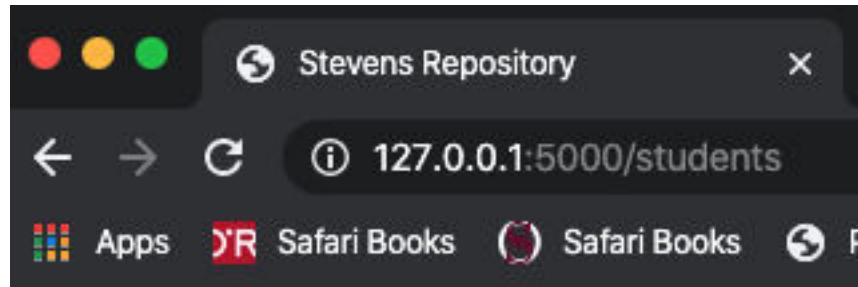
unchanged

```

buggy_static_table2.py > ...
9  @app.route('/students')
10 def students_summary() -> str:
11     data: List[Dict[str, str]] = [ # partial list of students
12         {'cwid': '10115',
13          'name': 'Bezos, J.',
14          'major': 'SFEN',
15          'complete': '2',
16        },
17         {'cwid': '10102',
18          'name': 'Gates, B.',
19          'major': 'CS',
20          'complete': '3',
21        },
22     ]
23
24     return render_template('student_courses.html',
25                           title='Stevens Repository',
26                           table_title='Student Summary',
27                           students=data)
```

The variable names  
match... Must be  
something else

Oops! The attribute  
names must match



## Stevens Repository

### Student Summary

Missing data field!

CWID	Name	Major	Completed Courses
10115	Bezos, J.	SFEN	2
10102		CS	3



# Running database queries from Python

Dish: \_\_\_\_\_

## Recipe

Serves: \_\_\_\_\_

1. import sqlite3
2. Specify the Sqlite database file name
3. Specify the query
4. Open a connection to the database
5. Execute a query
6. Fetch the results
7. Use the results in your Python program



# Display # completed courses by student

## Steps:

1. Write a query to pull the data
2. Create a template for the new web page
3. Update Stevens.py with a new function to return the HTML for the web page on our growing web site
  - a) Connect to the database
  - b) Run the query
  - c) Convert the results to a list of dicts
  - d) Render the new template and the database query results to Jinja2
  - e) Earn fame and fortune



You'll need to do this for the homework assignment!





# Query to compute completed courses

DataGrip

```
select s.cwid, s.name, s.major, count(*) as complete  
from students s join grades g on s.cwid=g.StudentCWID  
group by s.name  
order by s.name
```

Query

CWID	Name	Major	courses
10115	Bezos, J	SFEN	2
11714	Gates, B	CS	3
10103	Jobs, S	SFEN	2
10183	Musk, E	SFEN	2

Query Results



# Generate a new template

templates > student\_courses.html

```
1  {% extends 'base.html' %}  
2  {% block body %}  
3  <h2>{{ title }}</h2>  
4  <h3> {{ table_title }}</h3>  
5  <table border="1">  
6      <th>CWID</th>  
7      <th>Name</th>  
8      <th>Major</th>  
9      <th>Completed Courses</th>  
10     {% for student in students %}  
11         <tr>  
12             <td>{{ student.cwid }}</td>  
13             <td>{{ student.name }}</td>  
14             <td>{{ student.major }}</td>  
15             <td>{{ student.complete }}</td>  
16         </tr>  
17     {% endfor %}  
18 </table>  
19 {% endblock %}
```

Desired output in the browser

## Stevens Repository

### Number of completed courses by Student

CWID	Name	Major	Completed Courses
10115	Bezos, J	SFEN	2
11714	Gates, B	CS	3
10103	Jobs, S	SFEN	2
10183	Musk, E	SFEN	2

Reuse template  
from the static table



# Update completed\_table.py

```
completed_table.py > ...
1  """
2      Example of a dynamic table from Python with Flask
3  """
4  from flask import Flask, render_template
5  import sqlite3
6  from typing import Dict
7
8  DB_FILE: str = '/Users/jrr/Documents/Stevens/810/Assignments/HW11_Repository/HW11.sqlite'
9
10 app: Flask = Flask(__name__)
11
12 @app.route('/completed')
13 def completed_courses() -> str:
14     query = """select s.cwid, s.name, s.major, count(*) as complete
15             from students s join grades g on s.cwid=g.StudentCwid
16             group by s.name
17             order by s.name"""
18
19     db: sqlite3.Connection = sqlite3.connect(DB_FILE)
20
21     # convert the query results into a list of dictionaries to pass to the template
22     data: Dict[str, str] = \
23         [{cwid: cwid, name: name, major: major, complete: complete}
24          for cwid, name, major, complete in db.execute(query)]
25
26     db.close() # close the connection to close the database
27
28     return render_template('student_courses.html',
29                           title='Stevens Repository',
30                           table_title="Number of completed courses by Student",
31                           students=data)
32
33 app.run(debug=True)
```

Specify the query

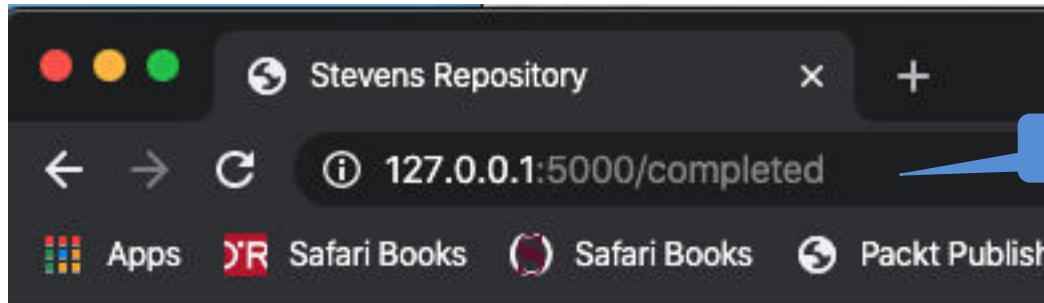
Connect to the database

Convert the query results into  
a list of dictionaries

Render the template



# Success! Display the results



## Stevens Repository

Titles are passed to the template

### Number of completed courses by Student

CWID	Name	Major	Completed Courses
10115	Bezos, J	SFEN	2
11714	Gates, B	CS	3
10103	Jobs, S	SFEN	2
10183	Musk, E	SFEN	2

Database query results presented on a web page



# Intro to HTML Forms

```
<form action="display" method="post">
    <!-- text input -->
    Name:
    <input type="text" name="name">
    <br><br>

    <!-- radio buttons -->
    <input type="radio" name="student" id="Student"> Student
    <input type="radio" name="instructor" id="Instructor"> Instructor
    <br><br>

    <!-- pull down list -->
    Major:
    <select name="major" id="">
        <option value="CS">Computer Science</option>
        <option value="ME">Mechanical Engineering</option>
        <option value="SSW">Software Engineering</option>
        <option value="SYEN">System Engineering</option>
    </select>
    <br><br>

    <input type="submit">
</form>
```

HTML

Rendered Form

Name:

Student  Instructor

Major:



# Populating HTML Forms with Flask

Use Flask and database queries to populate forms

```
<form action="display" method="post">
    <!-- text input -->
    Name:
    <input type="text" name="name">
    <br><br>

    <!-- radio buttons -->
    <input type="radio" name="student" id="Student"> Student
    <input type="radio" name="instructor" id="Instructor"> Instructor
    <br><br>

    <!-- pull down list -->
    Major:
    <select name="major" id="">
        <option value="CS">Computer Science</option>
        <option value="ME">Mechanical Engineering</option>
        <option value="SSW">Software Engineering</option>
        <option value="SYEN">System Engineering</option>
    </select>
    <br><br>

    <input type="submit">
</form>
```

Name:

Student  Instructor

Major:

Replace hard-coded values  
with computed values



# To Do:

Render a page with all student names

Ask the user to choose student

Display all courses and grades for that student

**Please choose a student**

Bezos, J



Submit

All student  
names from  
database

**Student Repository**

**Courses/Grades for CWID 10115**

Course	Grade
SSW 810	A
CS 546	F

Results for  
specified  
student



# Populating HTML Forms with Flask

student\_form.py – a Flask app runs a query to pull the student names and renders students.html to present a form

```
""" Example of using an HTML form with Flask and Python """
import sqlite3
from flask import Flask, render_template, request
from typing import Dict, Tuple

DB_FILE: str = '/Users/jrr/Documents/Stevens/810/Assignments/HW11_Repository/HW11.sqlite'
app = Flask(__name__)

@app.route('/choose_student')
def choose_student() -> str:
    query: str = "select cwd, name from students group by cwd, name"
    db: sqlite3.Connection = sqlite3.connect(DB_FILE)
    students: Dict[str, str] = [{'cwd': cwd, 'name': name} for cwd, name in db.execute(query)]
    db.close()
    return render_template('student_form.html', students=students)
```

Replace hard-coded values in the form with values from the database

Render a form with student names and allow user to choose

Retrieve all student names from database



# student\_form.html

## Display a pull-down menu with all students' names

templates > student\_form.html

```
1  {% extends 'base.html' %}                                Page to render on submit
2  {% block body %}
3      <h3>Please choose a student</h3>
4      <form action="show_student" method="post">
5          <!-- pull down list -->
6          <select name="cwid" id="">
7              {% for student in students %}
8                  <option value="{{ student.cwid }}">{{ student.name }}</option>
9              {% endfor %}
10         </select>
11         <br><br>
12
13         <input type="submit">
14     </form>
15  {% endblock %}
```

Annotations from right side:

- Add a drop down menu option for each student's name
- Return the CWID
- Display the student's name



# Retrieve the student courses/grades

student\_form.py > ...

```
17     @app.route('/show_student', methods=['POST'])
18     def show_student() -> str:
19         """ user chose a student from the form and now want information """
20         if request.method == 'POST':
21             cwid: str = request.form['cwid']  
                Retrieve the cwid selected on the form
22
23             query: str = "select course, grade from grades where studentCWID=?"
24             args: Tuple[str] = (cwid,)
25             table_title: str = f"Courses/Grades for CWID {cwid}"  
                Retrieve the courses and
26
27             db: sqlite3.Connection = sqlite3.connect(DB_FILE)
28             rows: List[Dict[str, str]] = \
29                 [{course: course, grade: grade} for course, grade in db.execute(query, args)]
30             db.close()
31
32             return render_template('grade_form.html', title="Student Repository",
33                                     table_title=table_title, rows=rows)
```

Page rendered as result of post  
request from form

Retrieve the courses and  
grades from database

Render page with the results



# grade\_form.html

templates > grade\_form.html

```
1  {% extends 'base.html' %}  
2  {% block body %}  
3      <h2>{{ title }}</h2>  
4      <h3> {{ table_title }}</h3>  
5      <table border="1">  
6          <th>Course</th>  
7          <th>Grade</th>  
8          {% for row in rows %}  
9              <tr>  
10                 <td>{{ row.course }}</td>  
11                 <td>{{ row.grade }}</td>  
12             </tr>  
13         {% endfor %}  
14     </table>  
15 {% endblock %}
```

Page rendered as result of post request from form

Display a table with the courses and grades passed in from Python



# Success!

Rendered a menu button with all student names

Asked the user to choose student

Displayed all courses and grades for that student

**Please choose a student**

**Submit**

All student  
names from  
database

**Student Repository**

**Courses/Grades for CWID 10115**

Course	Grade
SSW 810	A
CS 546	F

Results for  
specified  
student



# Learning more about Flask Extensions

Flask supports many extensions to add functionality

- **Flask-Ask** – create Amazon Echo apps easily
- **Flask-Login** – user session management, logging in/out
- **Flask-Mail** – send mail from Flask
- **Flask-MongoAlchemy** – MongoDB support
- **Flask-Oauth** – common login support
- **Flask-Principal** – user authentication management
- **Flask-SQLAlchemy** – support SQLAlchemy
- **Flask-Upserts** – upload and serve files
- **Flask-User** – User account management

See : <http://flask.pocoo.org/extensions/>



# Learning more

## Flask resources

- <http://flask.pocoo.org/>
- <http://flask.pocoo.org/docs/0.12/tutorial/introduction/>
- <https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

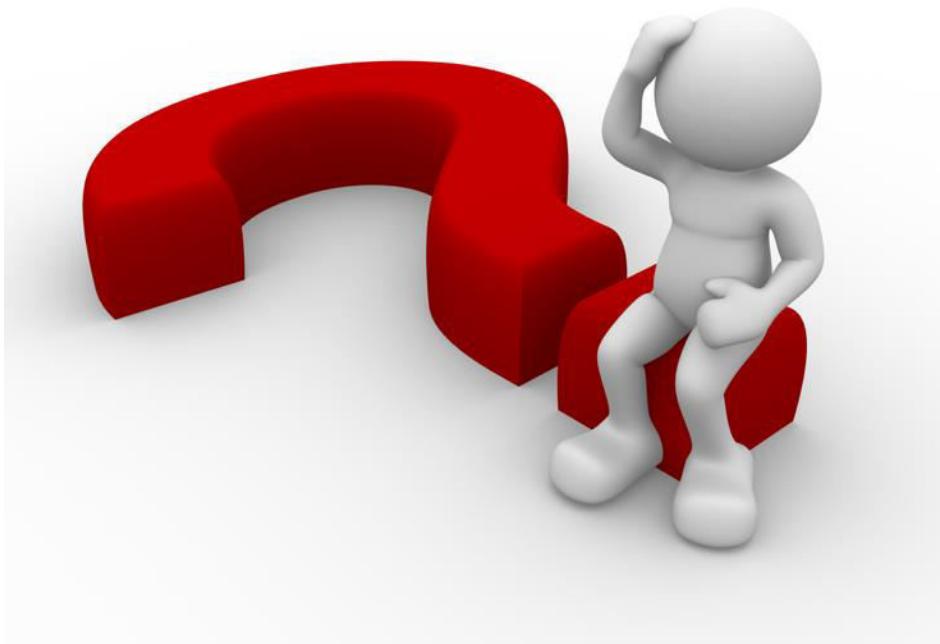
## Python and SQLite

- [http://sebastianraschka.com/Articles/2014\\_sqlite\\_in\\_python Tutorial.html](http://sebastianraschka.com/Articles/2014_sqlite_in_python Tutorial.html)

## Python and Databases

- <http://www.sqlalchemy.org/>

# Questions?





# SSW-810: Software Engineering Tools and Techniques

## NoSQL Database Overview

Prof. Jim Rowland  
Software Engineering  
School of Systems and Enterprises





# Acknowledgements

This lecture includes material from:

<http://www.tutorialspoint.com/articles/what-is-nosql-and-is-it-the-next-big-trend-in-databases>

<http://www.w3resource.com/mongodb/nosql.php>

<http://api.mongodb.com/python/current/index.html>

<https://www.tutorialspoint.com/mongodb/index.htm>

<https://docs.mongodb.com/manual/>

# Today's topics

## Alternatives to Relational Databases

NoSQL databases

MongoDB

Compass

PyMongo





# CRUD

All persistent storage solutions must support CRUD operations

- C CREATE
- R READ
- U UPDATE
- D DELETE



# RDBMS

Relational database technology is a great solution for many applications

Characteristics of good applications for RDBMS solutions:

Structured data

Data format can be defined in advance

Up to a few TBs of data

High availability transactions

High consistency  
Transactions

**ORACLE®**

The SQLite logo consists of a blue feather quill pen icon positioned above the word "SQLite" in a blue serif font.

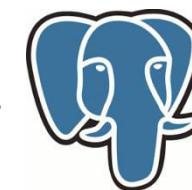
**SQLite**

The MySQL logo features a stylized blue dolphin leaping out of water to the left of the word "MySQL" in a blue and orange sans-serif font.

**MySQL™**



Microsoft®  
**SQL Server**®



**PostgreSQL**  
the world's most advanced open source database

# RDBMS Alternatives

Big Data applications created a need for new solutions

Many applications involve large data sets (Exabytes)

E.g. imagine Google searching 30 trillion web pages 100 billion times a month from a MySQL database



Source: <http://venturebeat.com/2013/03/01/how-google-searches-30-trillion-web-pages-100-billion-times-a-month/>

# NoSQL

## Unstructured data vs structured relational data

NoSQL supports structured, semi-structured and unstructured data

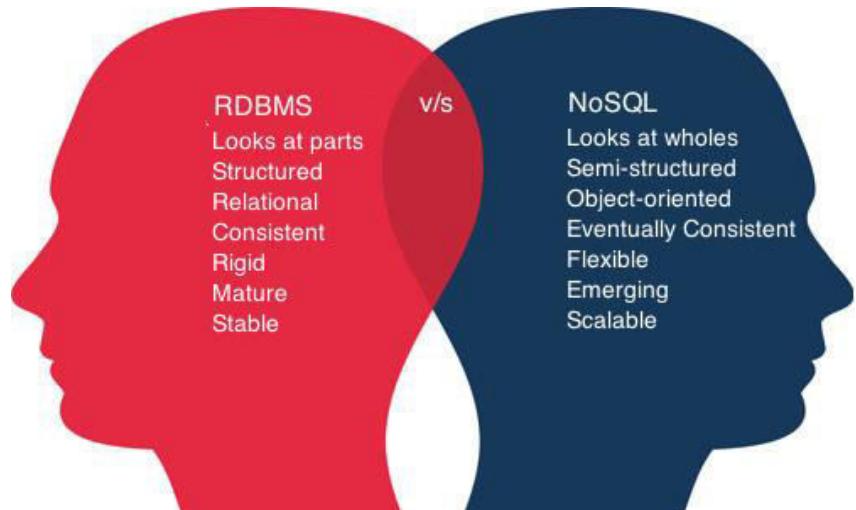
Replace relational tables with flexible structures

Spread the data across clusters of computers

NoSQL solutions are schema-less

Priorities:

- High performance
- High availability
- Scalability

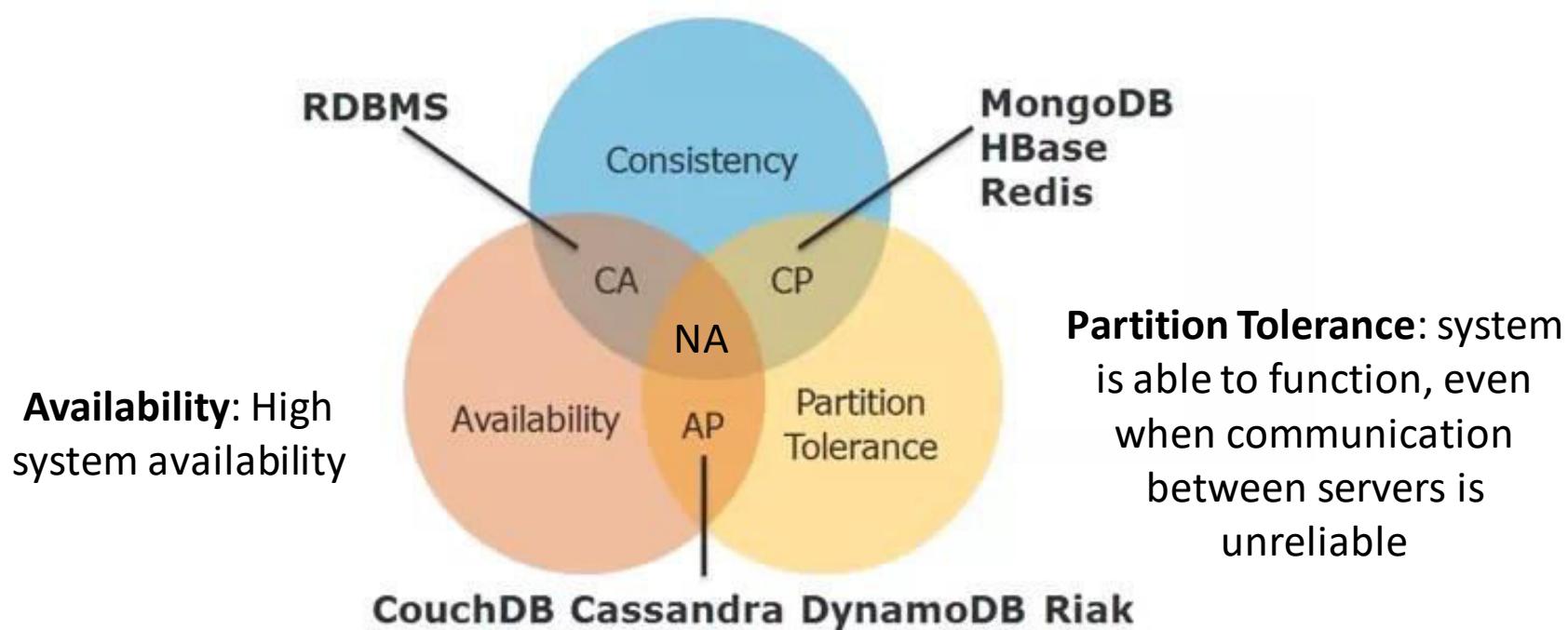


<http://www.smartdatacollective.com/sites/smardatacollective.com/files/RDBMSvsNoSQL.jpeg>

# CAP Theorem (Brewer's Theorem)

Distributed solutions require tradeoffs decided by business needs  
Can't have all three simultaneously

**Consistency:** Data is consistent across clients after an operation



<http://www.w3resource.com/mongodb/nosql.php>



# Four NoSQL data models

- 1. Key Value, e.g. Redis
  - 2. Document oriented, e.g. MongoDB
  - 3. Column based, e.g. CouchDB
  - 4. Graph based, e.g. Neo4j
- } Aggregate Objects      } Relationships

Not Only SQL

# Key/Value NoSQL databases

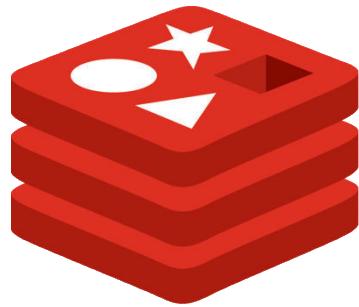
The key of the key/value pair is a unique value

Use the key to quickly access the data

Data may be a document, image, value, etc.

Some Key/Value solutions keep the data in memory for speed

Appropriate for read-mostly, read-intensive, large data repositories



redis



# Document oriented NoSQL databases

Store the entire document of key/value pairs

Typically JSON documents

”File” the document in the database

Use key to access the document

Store many types of document in a single collection

Store any kind or amount of data

No predefined schema

Very flexible to facilitate changes



# Column Based NoSQL databases

Focus on column-oriented data

All values of a column are stored together in files

Very space efficient

Don't store NULL values in the column

Better compression performance

Very efficient aggregation, e.g. sum, count, max, min, avg

Works well for data warehouses, BI, CRM, ...



# Row Store vs Column Store

## row-store



- + easy to add/modify a record
- might read in unnecessary data

## column-store



- + only need to read in relevant data
- tuple writes require multiple accesses

=>suitable for read-mostly, read-intensive, large data repositories

Source: <http://www.w3resource.com/mongodb/nosql.php>



# Graph based NoSQL databases

Store data with nodes, edges, and properties

- Nodes (vertices) represent entities

- Edges represent relationships

- Edges link ordered pairs of nodes

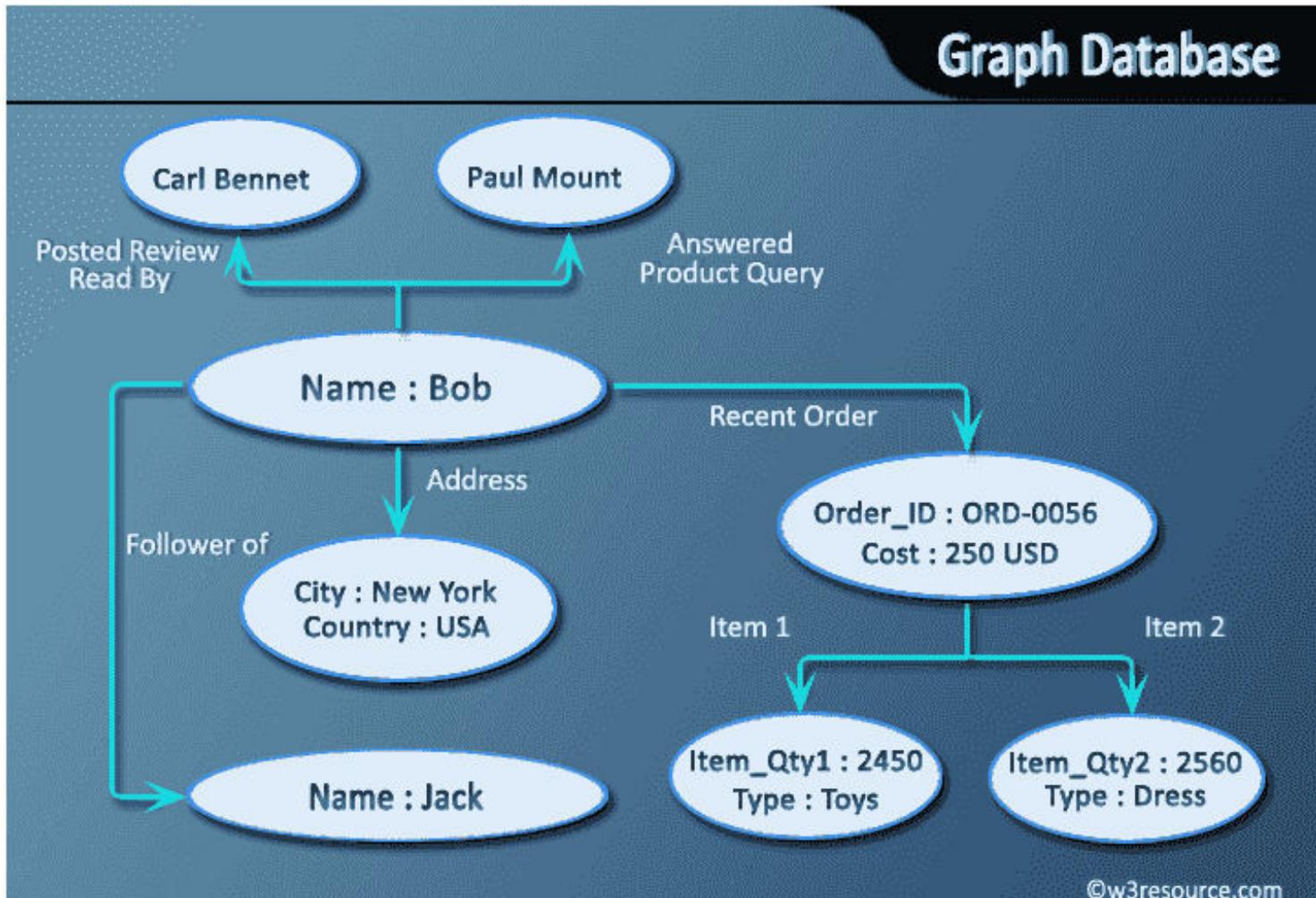
- Each node and edge has a unique identifier

Index of nodes and edges for fast lookup

Easy to explore relationships in the data



# Graph Database



Source: <http://www.w3resource.com/mongodb/nosql.php>

# MongoDB

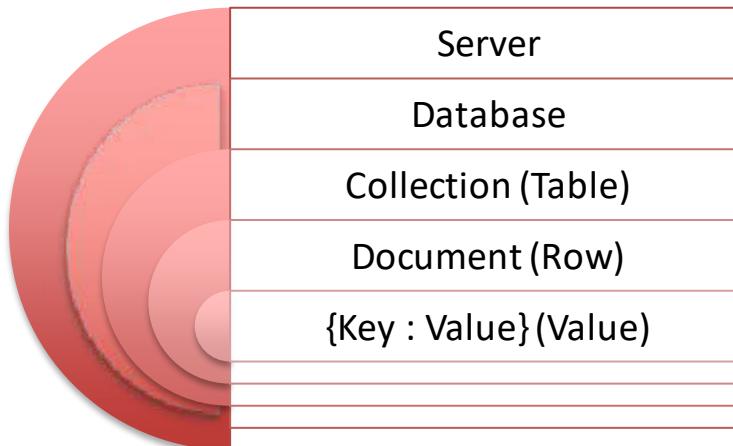
## Collections and Documents

Documents are stored as JSON-like objects

High performance and scalability

Optimized for scalable, distributed compute clusters

Provides data replication for reliability





# MongoDB Features

Data stored as JSON documents

```
{"Cwid": 123,  
 "Name": "Wright, D",  
 "Major": "SFEN",  
 "Grades": {"SSW 540": "A", "SSW 533": "B"} }
```

No fixed schema as in RDBMS

Each document may have different attributes and types

Add, rename, or remove attributes easily at any time



# MongoDB Features (CRUD)

Queries return all documents matching the query

{“Major” : “SFEN” }

Return all documents containing “Major” == “SFEN”

Display a subset of document fields

Sort results

Aggregations: \$sum, \$min, \$max,

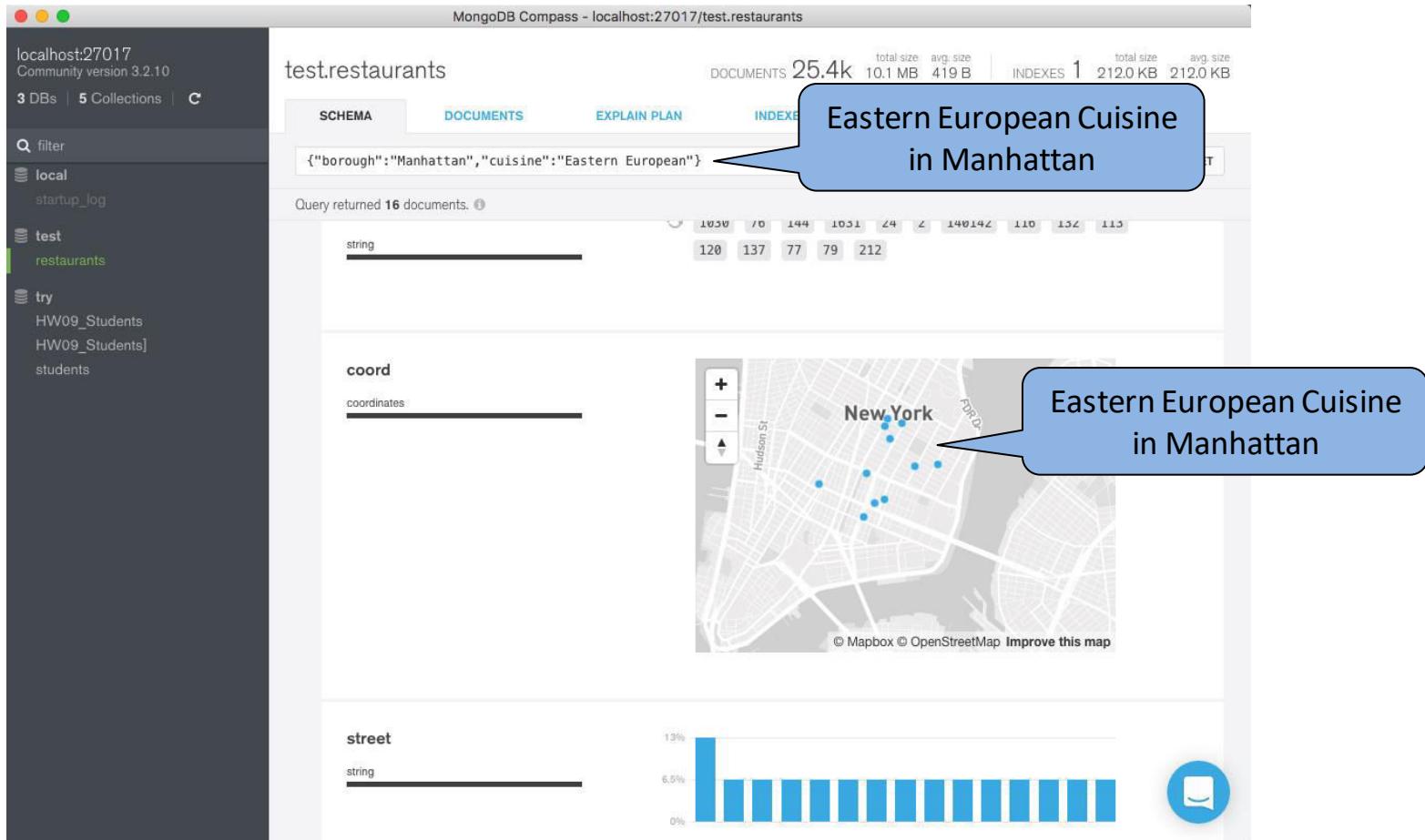
Update documents

Insert documents

Remove documents

# MongoDB Canvas

## Explore MongoDB databases easily



The screenshot shows the MongoDB Compass interface connected to a local MongoDB instance at localhost:27017. The database selected is 'test', and the collection is 'restaurants'. A search query is entered in the search bar: `{"borough": "Manhattan", "cuisine": "Eastern European"}`. The results show 16 documents found. Below the search bar, there are three input fields: 'string' (with value 'coordinates'), 'coord' (with value 'coordinates'), and 'street' (with value 'string'). To the right of these fields is a map of New York City, specifically Manhattan, with several blue dots representing the locations of restaurants. Two blue callout bubbles point from the text 'Eastern European Cuisine in Manhattan' to the search bar and the map respectively. At the bottom right of the interface is a blue circular icon with a white compass symbol.

MongoDB Compass - localhost:27017/test.restaurants

localhost:27017  
Community version 3.2.10

3 DBs | 5 Collections | C

filter

local

startup\_log

test

restaurants

try

HW09\_Students

HW09\_Students]

students

test.restaurants

DOCUMENTS 25.4k total size 10.1 MB avg. size 419 B INDEXES 1 total size 212.0 KB avg. size 212.0 KB

SCHEMA DOCUMENTS EXPLAIN PLAN INDEXES

`{"borough": "Manhattan", "cuisine": "Eastern European"}`

Query returned 16 documents. ⓘ

string

coord

coordinates

street

coordinates

New York

Hudson St

© Mapbox © OpenStreetMap Improve this map

13%  
6.5%  
0%

Eastern European Cuisine in Manhattan

Eastern European Cuisine in Manhattan

# MongoDB Canvas

## Explore MongoDB databases easily

MongoDB Compass - localhost:27017/test.restaurants

localhost:27017  
Community version 3.2.10  
3 DBs | 5 Collections | C

test.restaurants

DOCUMENTS 25.4k total size 10.1 MB avg. size 419 B INDEXES 1 total size 212.0 KB avg. size 212.0 KB

`{"borough": "Manhattan", "cuisine": "Eastern European"}`

Query returned 16 documents. + INSERT

`_id: ObjectId('580bfe1b0a0bfeb73917af10')`  
 ▶ address:object  
 borough: "Manhattan"  
 cuisine: "Eastern European"  
 ▶ grades:Array[6]  
 name: "Veselka Restaurant"  
 restaurant\_id: "40384528"

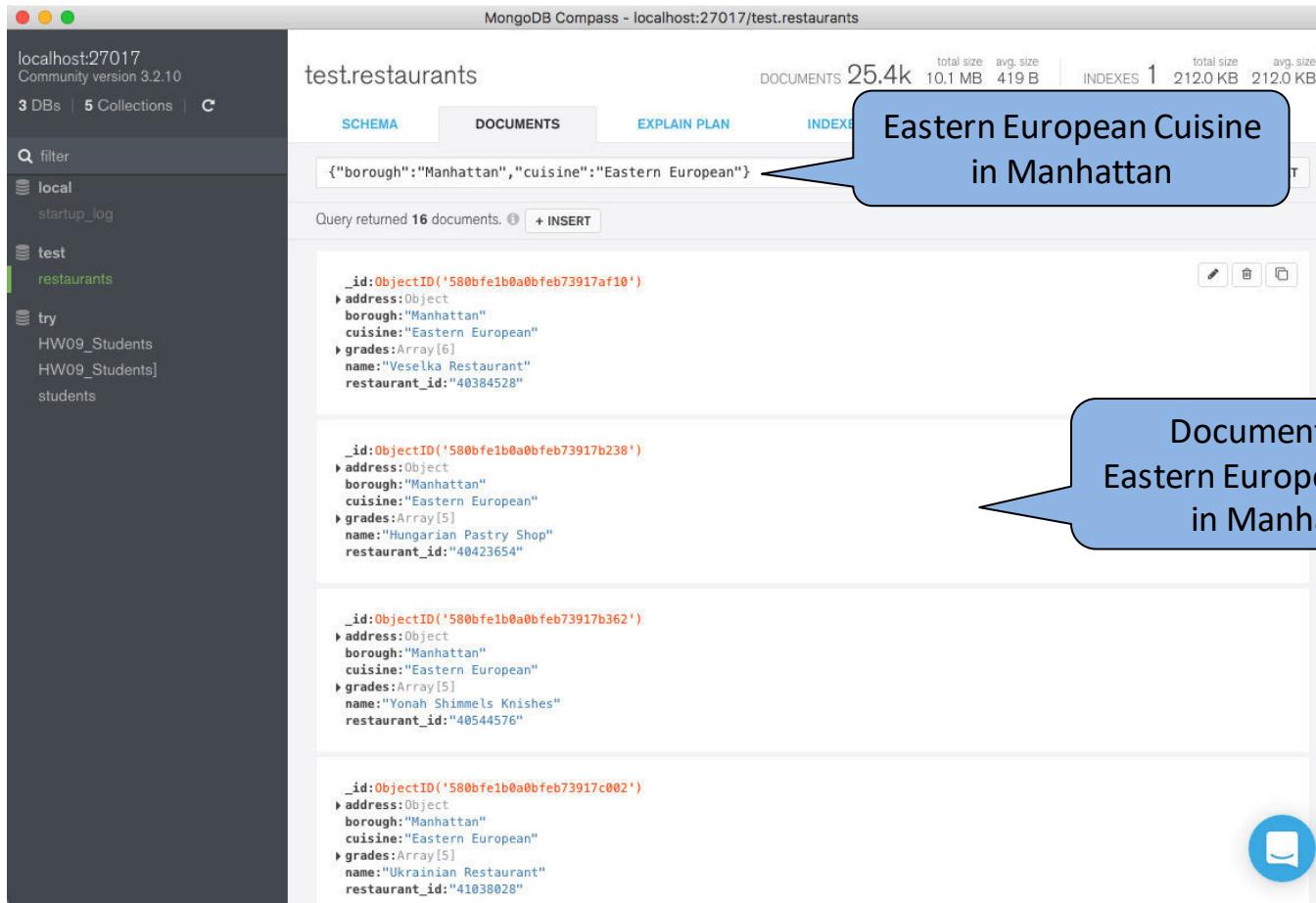
`_id: ObjectId('580bfe1b0a0bfeb73917b238')`  
 ▶ address:object  
 borough: "Manhattan"  
 cuisine: "Eastern European"  
 ▶ grades:Array[5]  
 name: "Hungarian Pastry Shop"  
 restaurant\_id: "40423654"

`_id: ObjectId('580bfe1b0a0bfeb73917b362')`  
 ▶ address:object  
 borough: "Manhattan"  
 cuisine: "Eastern European"  
 ▶ grades:Array[5]  
 name: "Yonah Schimmel's Knishes"  
 restaurant\_id: "40544576"

`_id: ObjectId('580bfe1b0a0bfeb73917c002')`  
 ▶ address:object  
 borough: "Manhattan"  
 cuisine: "Eastern European"  
 ▶ grades:Array[5]  
 name: "Ukrainian Restaurant"  
 restaurant\_id: "41038028"

Eastern European Cuisine in Manhattan

Documents with Eastern European Cuisine in Manhattan





# Accessing MongoDB from Python

PyMongo provides a powerful interface to MongoDB

```
from pymongo import MongoClient

client = MongoClient()
db = client['test']
collection = db['restaurants']

for restaurant in collection.find(
    {"borough": "Manhattan",
     "cuisine": "Eastern European"}):
    print(restaurant['name'])
```

---

PyMongo Tutorial:

<http://api.mongodb.com/python/current/index.html>



# Getting Started with MongoDB

The screenshot shows the MongoDB Documentation homepage at <https://docs.mongodb.com>. A blue speech bubble highlights the "Getting Started Documentation" section, which contains a note about a DDOS attack. Another blue speech bubble highlights the "MongoDB Server" section. A third blue speech bubble highlights the "Compass GUI" section, which is part of the "MongoDB Cloud Services" row. The page also features sections for MongoDB Atlas, MongoDB Ops Manager, MongoDB Drivers, MongoDB Compass, MongoDB Spark Connector, and MongoDB BI Connector.

https://docs.mongodb.com

Getting Started Documentation

MongoDB Server

Compass GUI

MongoDB Documentation

You may have experienced difficulty reaching the MongoDB Documentation due to a large-scale DDOS attack affecting many sites in the US and elsewhere. See the [Dyn status page](#) for details. We apologize for any inconvenience.

MongoDB Server

MongoDB Server is an open-source, document database designed for ease of development and scaling.

MongoDB Atlas

MongoDB Atlas as a service for experts who engineer.

MongoDB Cloud Manager

MongoDB Ops Manager

MongoDB Ops Manager is an enterprise operations management solution for MongoDB.

MongoDB Drivers

MongoDB Drivers allow you to work with MongoDB databases from your favorite programming languages.

MongoDB Compass

MongoDB Compass is a visual data exploration and analysis tool.

MongoDB Spark Connector

MongoDB Connector for Spark provides integration between MongoDB and Apache Spark.

MongoDB BI Connector

MongoDB Connector for Business Intelligence connects business intelligence tools to your MongoDB databases.



# NoSQL Summary

NoSQL database solutions offer alternatives to RDBMS

More flexible

More scalable to support very large datasets

Interesting alternative to RDBMS

# Questions?

