W200 - Intro to Python

Functions

5

Python for Data Science: Fall 2018						All due dates are tentative and may be changed by instructors. Homework due dates are 11:59pm PST the night before live session.					
Mon	**Tues**	**Weds**	**Thurs**	**Async Unit**	**Sync Week**	**Async to Review (Prior to Class)**	**Projects (20% each)**	**Exams (10% each)**	**HW Assigned (30% total)**	**HW Due**	**Notes**
Sep 3	Sep 4	Sep 5	Sep 6	1	1	Introduction to Programming, the Command Line, and Source Control			unit 1		A make-up class will be scheduled for Monday class.*
Sep 6									unit 1		[This is the make-up for Monday 4 pm session.]
Sep 10	Sep 11	Sep 12	Sep 13	2	2	Starting Out with Python			unit 2	unit 1	
Sep 17	Sep 18	Sep 19	Sep 20	3	3	Sequence Types and Dictionaries			unit 3	unit 2	9/17/2018 - Last day to add or drop a class
Sep 24	Sep 25	Sep 26	Sep 27	4	4	More About Control and Algorithms			unit 4	unit 3	
Oct 1	Oct 2	Oct 3	Oct 4	5	5	Functions			unit 5	unit 4	
Oct 8	Oct 9	Oct 10	Oct 11	6	6	Complexity	Project 1 Assigned		scrabble	unit 5	
Oct 15	Oct 16	Oct 17	Oct 18	7	7	Classes			unit 7	scrabble	
Oct 22	Oct 23	Oct 24	Oct 25	8	8	Object-Oriented Programming/<text a	Project 1 Final Proposal Due	Exam 1 Start	x	unit 7	
Oct 29	Oct 30	Oct 31	Nov 1	9	9	Working With Text and Binary Data/numpy		Exam 1 Due	x		
Nov 5 - 9 Fall Break & Immersion											
Nov 12	Nov 13	Nov 14	Nov 15	10	10	NumPy	Project 1 Presentations		unit 9 / HW10		A make-up class will be scheduled for Monday Class
Nov 19	Nov 20	Nov 21	**Nov 22**	11	11	Data Analysis With Pandas	Project 2 Assigned		unit 10 / HW11	unit 9 / HW10	A make-up class will be scheduled for the Thursday Classes
Nov 26	Nov 27	Nov 28	Nov 29	12	12	Plotting and Visualization	Project 2 Proposal Due		unit 11 / HW12	unit 10 / HW11	
Dec 3	Dec 4	Dec 5	Dec 6	13	13	Pandas Aggregation and Group Operations		Exam 2 Start	x	unit 11 / HW12	
Dec 10	Dec 11	Dec 12	Dec 13	14	14	Testing	Project 2 Presentations!	Exam 2 Due	x		Last Day of Class. bring beer Congratulations!
Last Day of Instruction - December 14											

https://docs.google.com/spreadsheets/d/1sVV7-4OHZ-EDNqkMJ55OPUfz_QLJ4LZNuZl-cRaxgV0/edit#gid=0

Today …

Homework 4:

- for loop
- chess
- algorithms - binary search (discussion & updates)
- comprehensions.

Discuss: what was the hardest part of homework 4? How much time did you spend on this week's assignment?

Homeworks 3 & 4

- Week 3:
  - Pig Latin
  - Matrix Inverter
  - To-do List
  - Fibonacci Series
  - Pascal's Triangle
- Week 4:
  - for loop
  - chess board
  - algorithms - binary
  - comprehensions
  - How long did it take to complete assignments?

Notes about homeworks

- Remember to use either all tabs or 4 spaces [not both]
- The usual style is to have a space before/after operands
  - (e.g., `a = b + c`, not `a=b+c`)
- Comments are encouraged - please put 'em on their own line, as opposed to appending at the end of the line.
- When submitting the final version of HW, please delete any personal debugging statements. Thanks.

git hub | branching & merging

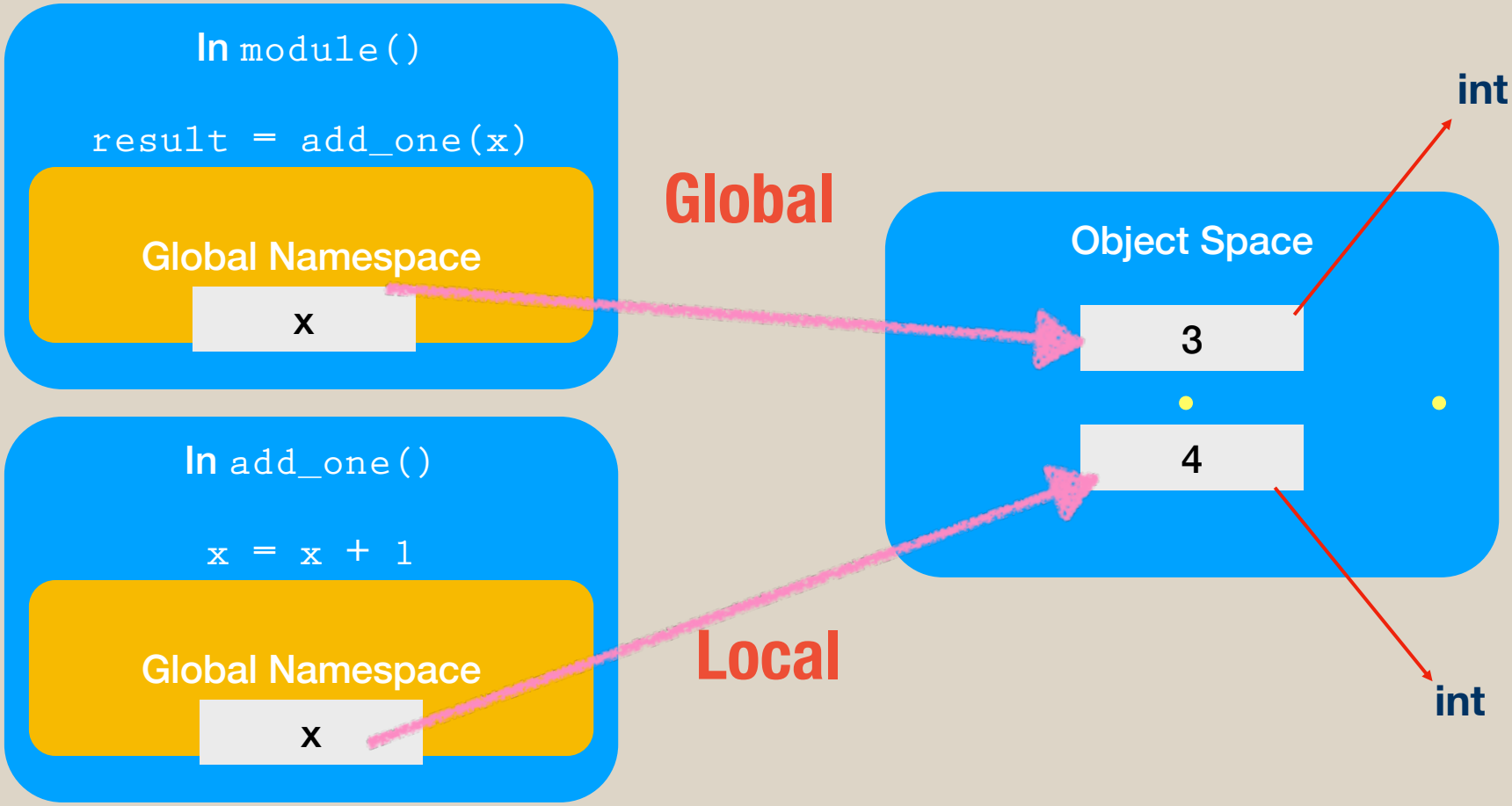
- Make the branch:

```
git checkout -b <name>      # to add a new branch

git branch                  # tells you what branches there are.
```

- Merge into the branch:

```
git checkout master         # change to branch you want to merge

git merge <alt branch>      # "fast forward" merge indicates no conflicts.
```

Name Space | Global v. Local



Namespace | Global v. local

- Discussion: What have separate namespaces?
  - What are local variables used for? [examples?]
  - What are global vars used for? [examples?]
  - What happens when a var is called but not defined in the local namespace?

Functions | Anatomy

- Vocabulary:

1. `myfunction(x)` accepts “arguments”; “parameters” inside
2. docstring # help text [readability, reusability]
3. code reuse; modularity; abstraction
4. functions are objects
5. functions are not executed until called by the script
6. Not all functions return a value, but if they do … look for `return`

```
## square root algorithm as a function

def sqrt(x, epsilon):
    """Newton's method to find square root with precision
       epsilon (Heron's algorithm)"""

    ans = 1
    num_guesses = 0
    while abs(x/ans - ans) > epsilon:
        ans = (x/ans + ans)/2
        num_guesses += 1
    return ans
```

1 define the function

3 arguments become parameters inside function

```
def distance_to_origin(x,y):
    """find the distance from point at (x,y) to the origin"""
    ans = sqrt(x**2 + y**2, 0.00001)
    return ans

magnitude = distance_to_origin(x,y)
```

4 internal operations

5 return variable

2 execute function with parameters

6 assignment to var outside of the function

This may be already clear to you. Thinking about efficiencies, reflect on the number of steps involved from the computer's p.o.v.

Functions as objects

- remember that unlike most computer languages Python sees everything as an object (even if our other experience makes us question the syntax).
- And interestingly we can query Python to query itself!
  - It has a type

```
>>> type(round10)
>>> function
```

```
a = round10
 a(12)
 10
```

  - And we can bind it to a variable name. Can be passed as an argument [perform operation (function) on iterable (example: grade_list)]:

```
def apply_to_grades(operation, grade_list):
    return [(name, operation(grade)) for name, grade in grade_list]

grade_list = [("Betty", 88), ("Steve", 75), ("Bob", 73), ("Ming Wa", 75)]
print( apply_to_grades(round10, grade_list) )
```

Functions | map & lambda

- map() `list(map(round10, (23,24,42,66)))` `[20, 50, 40, 70]`
  - Apply a function to each element of an iterable …
- lambda functions ("anonymous")

```
lambda x : 100 - (100 - x)/2
<function __main__.<lambda>>
```

  - The functions aren't named
  - We can pass the lambda as an argument

```
apply_to_grades(lambda x : 100 - (100 - x)/2, grade_list)
 [('Betty', 94.0),
  ('Steve', 87.,5),
  ('Bob', 86.4),
  ('Ming Wa', 97.0)]
```

map()

- map() executes a specified function for each in an iterable. The item is sent to the function as a parameter:
  - `map(function, iterables)`

Calculate the length of each word in the tuple:

```
def myfunction(n):
    return len(n)

x = map(myfunction, ('Boston', 'Back Bay', 'Cambridge'))
print(list(x))
```

Outputs: [6, 8, 9]

Make new combinations by providing 2 iterable objects:

```
def myfunction(a, b):
    return a + b

x = map(myfunction, ('a', 'b', 'c'), ('1', '2', '3'))
# convert the map into a list for readability:
print(list(x))
```

Outputs: ['a1', 'b2', 'c3']

lambda

- A lambda function is a small, anonymous function, taking any number of arguments but can have only one expression:
  - `lambda arguments : expression`

Lambda that adds 10 to number passed and prints result:

```
x = lambda a : a + 10
print(x(5))
```

Outputs: 15

Lambda that sums argument a, b, c and prints result:

```
# Technically its not anonymous if you name it so …
(lambda a, b, c : a + b + c) (5, 6, 2)
```

Outputs: 13

Lambda is usually passed as an argument:

```
apply_to_grades(lambda x : 100 - (100 - x)/2, grade_list)
```

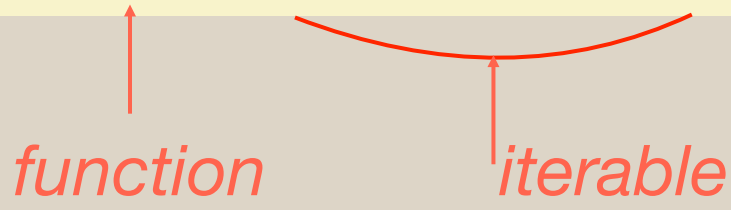
Outputs:

```
[('Betty', 94.0),
 ('Steve', 87.,5),
 ('Bob', 86.4),
 ('Ming Wa', 97.0)]
```

Functions | map & lambda

- Common to combine both `map` & `lambda`
- This uses lambda **as an argument** and repeats it's use

```
list(map(lambda x: x**2, (23, 25, 52, 66)))
```



Outputs: `[529, 2025, 1764, 4356]`

```
list(map((lambda a, b : a + b), [5, 50] , [6, 60]))
```

Outputs: `[11, 110]`

Special Arguments | flexibility

- What does it mean for code to be “brittle” or “fragile”?

- Special methods increase flexibility [default types, None values]

```
def feedback(grade=None, comment=None):
    text = “” if comment == None else “ - “ + comment

    if grade == None:
        return(“Grade is missing. “ + text)
    elif grade >= 90 and grade <= 100:
        return (“A “ + text)
    elif grade >= 80 and grade < 90:
        return(“B “ + text)
    elif grade >= 70 and grade < 80:
        return(“C “ + text)
    elif grade >= 60 and grade < 70:
        return(“D “ + text)
    else:
        return(“F “ + text)
```

Outputs:

```
print(feedback(80))
B

print(feedback(75, ‘Please study more’))
C - Please study more

print()
Grade is missing.
```

Default values | modified over time

- Default values won't be reset, if called a second time!
- "Permanent" attributes of a function [in the global namespace]
- Once used, modified:

```
def add_total(order_list = []):
    total = sum([quantity for name, quantity in order_list])
    order_list.append( ("Total", total) )
    print(order_list)
```

Outputs:

```
add_total()
[('Total', 0)]

add_total()
[('Total', 0), ('Total', 0)]
```

Default args | make it clear

- Arguments usually use positional cues …
  - We can define in any order if we specify keywords…

```
print(feedback(90, comment="Keep it up!"))
```

Outputs: `A - Keep it up!`

```
print(feedback(comment="Not bad.", grade=88))
```

Outputs: `B - Not bad.`

Arguments

- So far, just talked about arguments we've passed in Python … but … lots of need for passing parameters from the Operating System (O/S) … the "argument vector", usually `args[0, 1, 2]`

- Accessible through the library that lets python communicate with the O/S (`import sys`).

```
$ python
>>> myProgram("hello")
```

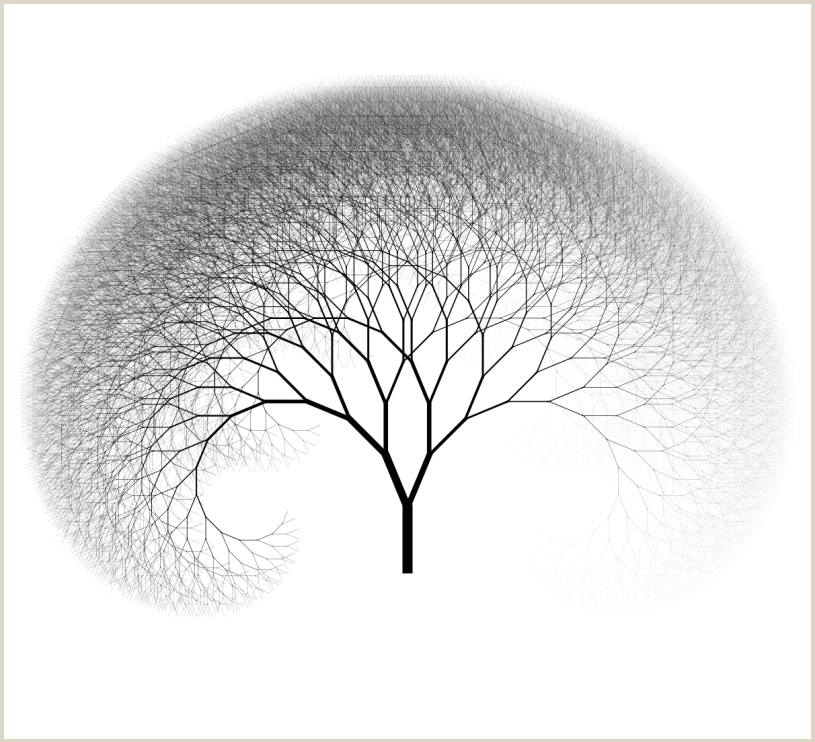
```
myProgram
…
main(args[]
   print(args[0])
```

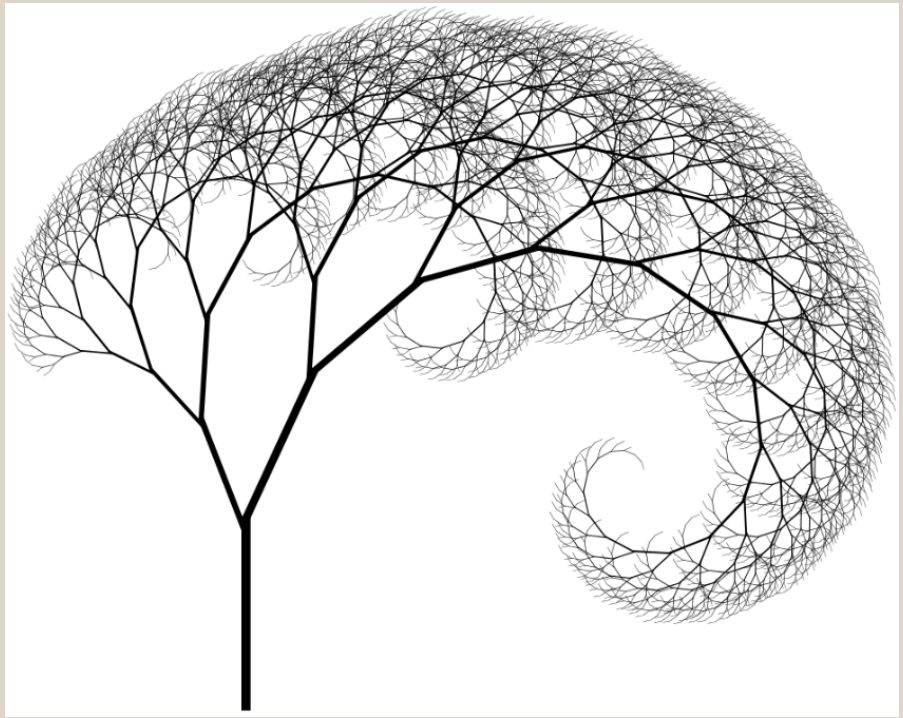
Demo script in python:

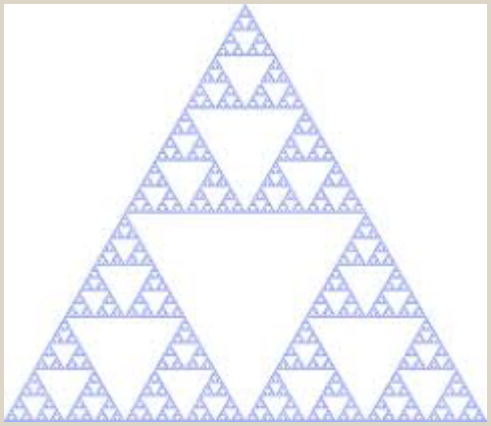
```
import sys

print(sys.argv)

if len(sys.argv) > 1:
    name = sys.argv[1]
else:
    name = input("Enter your name: ")

for i in range(len(name), 0, -1):
    print( name[0:i], end = " ")
    for j in range(i, len(name)):
        print(" " * (j-1) + name[j], end = "")
    print("")
```

Outputs:

```
>>> import sys
>>> print(sys.argv)
['']
>>> if len(sys.argv) > 1:
...     name = sys.argv[1]
... else:
...     name = input("Enter your name: ")
...
Enter your name: Fifi
>>> for i in range(len(name), 0, -1):
...     print(name[0:i], end = " ")
...     for j in range(i, len(name)):
...         print(" " * (j-1) + name[j], end = "")
...     print("")
...
Fifi
Fif   i
Fi  f  i
F i f  i
>>>
```



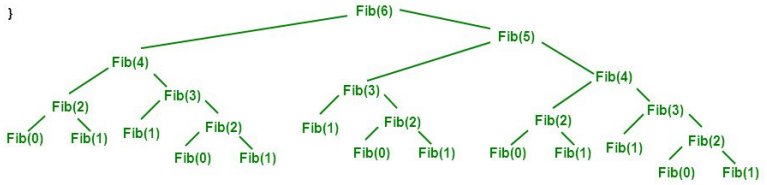


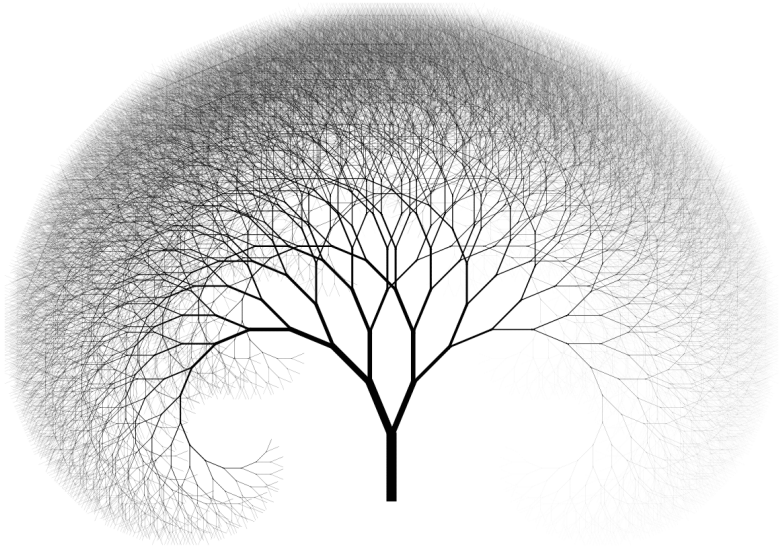


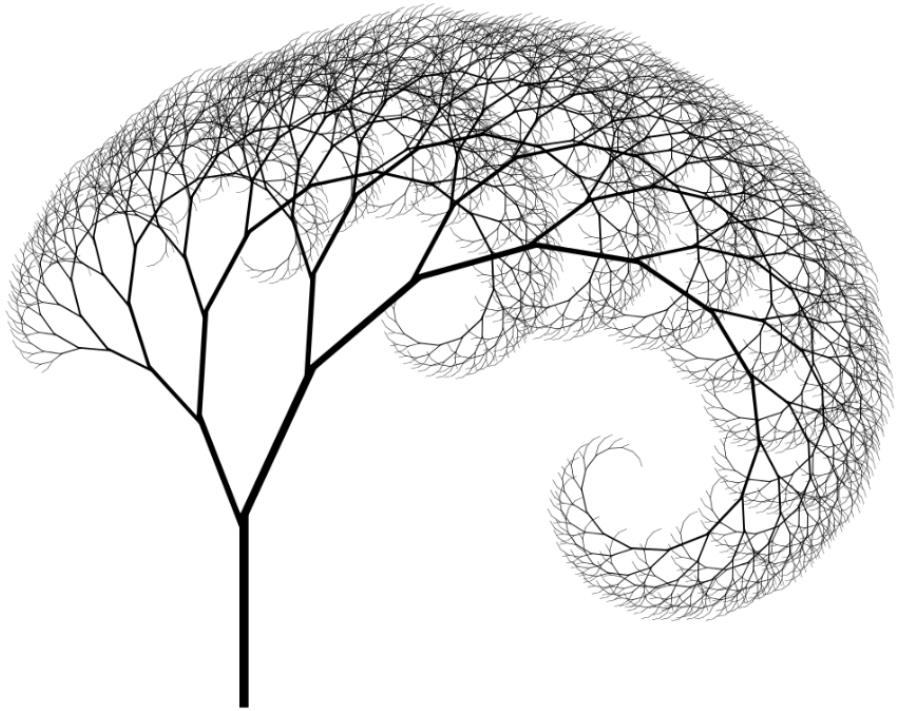
Excessive Recursion: Example

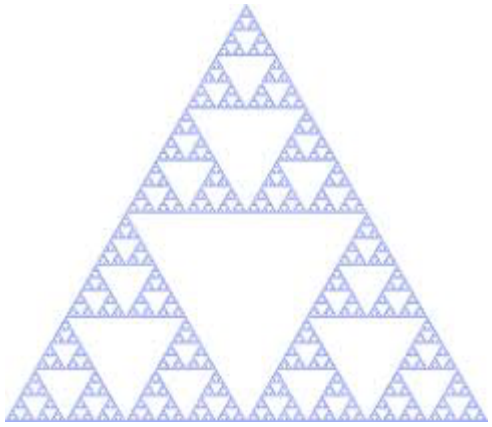
- To show how much this formula is inefficient, let us try to see how Fib(6) is evaluated.

```
int fib(int n) {
  if (n<2)
    return n;
  else
    return fib(n-2)+Fib(n-1);
}
```





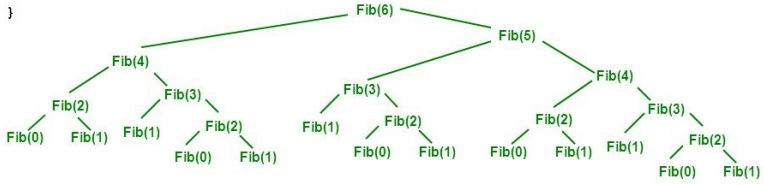




Excessive Recursion: Example

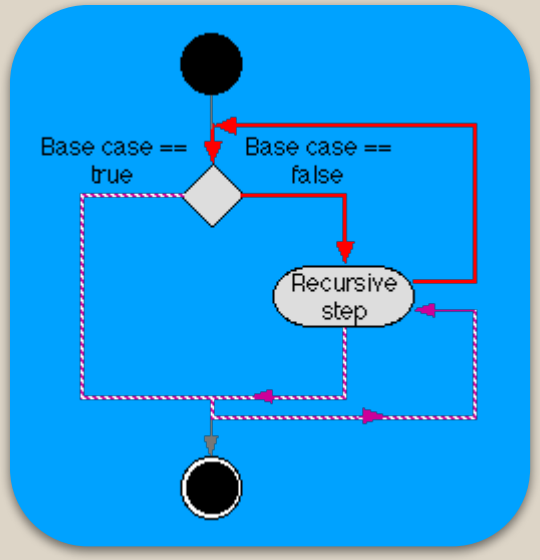
- To show how much this formula is inefficient, let us try to see how Fib(6) is evaluated.

```
int fib(int n) {
  if (n<2)
    return n;
  else
    return fib(n-2)+Fib(n-1);
}
```



Recursion

What is recursion?



Recursion occurs a function calls itself.

To use recursion, we must specify:

1) a base case and
2) a recursive rule.

The "base case" ends the process of the recursive rule (loop) calling itself.

Recursion

- Identify the “base case” and the “recursive rule”. Why do we need each?

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

Recursion

- Identify the "base case" and the "recursive rule". Why do we need each?

Base case

Recursive rule

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

One reason to think about recursion is for algorithm efficiency. Calling the same algorithm can increase operating time $O(n) + xn^2$ where n is the number of recursions and x is the # of steps.

Recursion

- *Question:* why use recursion instead of a loop?

Recursion allows us to know "even less" about the structure of a problem. E.g., we can traverse interesting data structures, such as trees and .json, without knowing about them in advance.

We've discussed that a "while" loop allows us to run a single loop an unknown number of times.

Recursion allows us to run an unknown number of loops, an unknown number of times.

"Do or do not! There is no try!"

[Is Yoda anticipating try … except block!?]

Errors | Checking & failing gracefully

- Do it all, or don't do any of it. And prepare for any type of error. Our buddies try … except.

```
try:
    x = float(input("Enter a number: "))
    print("The reciprocal is ", 1/x)
except ValueError:
    print("Sorry, your input was not valid.")
except ZeroDivisionError:
    print("Sorry, zero doesn't have a reciprocal")
except:
    Print "Yikes, something else is wrong."
```

specific

general

Notice we have both "general" and "specific" exceptions. Very useful for checking SQL, file access, insertion, user/file input, end-of-file (EOF), file not found (FNF), etc.!

Errors … "as e"

- Programming languages that support `try` … generate an error as a class, an "exception" class. This class has methods to extract the type of error.
- When the exception (e) is raised, we capture it and convert the error (e) to a string [for us humans (grin)]…

1

```
try:
    sell("oranges", 15, inventory)
except Exception as e:
    print("Sorry, not enough to sell. " + str(e))
```

2

```
def sell(item, quantity, inventory):

    if item not in inventory:
        raise Exception(str(item) + " does not appear in inventory")
    q = inventory[item]

    if q < quantity:
        raise Exception("Sorry, not enough to sell.")
    inventory[item] = q - quantity
```

btw …

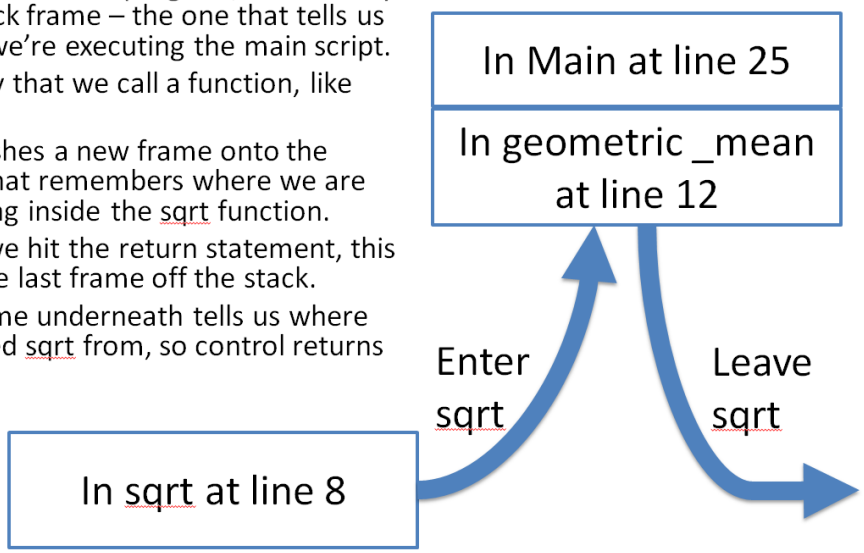
- if you know other languages, you'll have encountered try … catch. This is the same thing, with a different syntax.

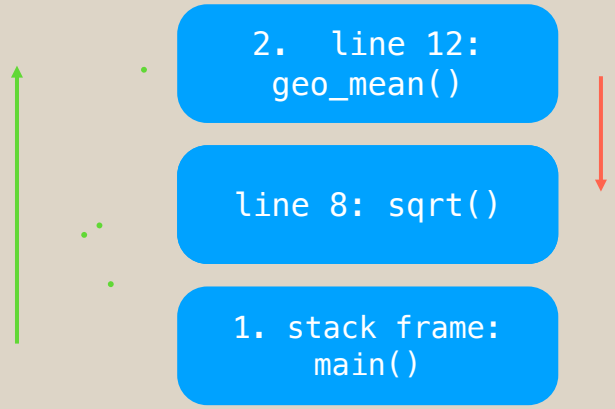
```
class myClass extends File throws Exception

try {
    fh = read("myfile.txt");
} catch(Exception e) {
    system.out.println("Error: "+e.getMessage())
}
```

Stack Trace | Visual understanding

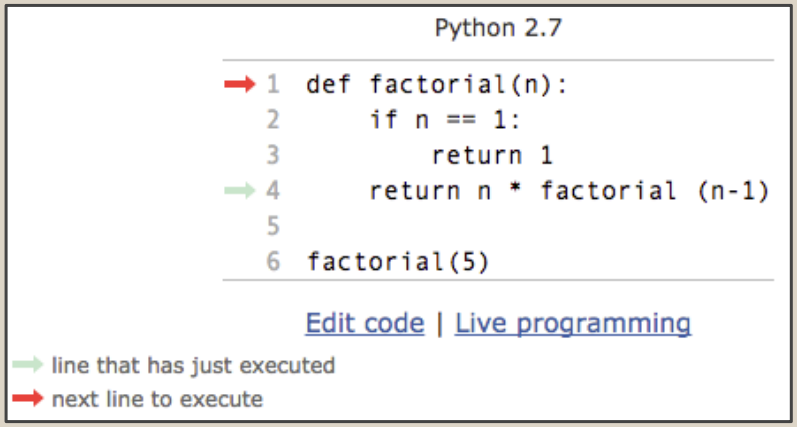
Call Stack

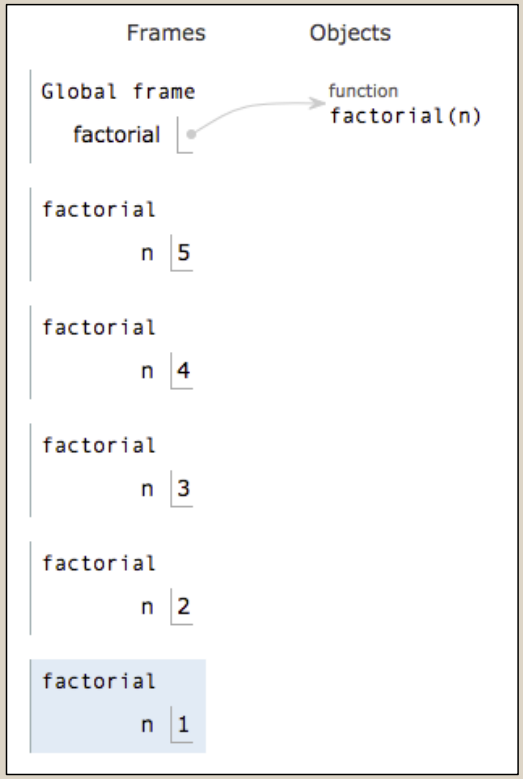
- When you start a program, there's only one stack frame – the one that tells us where we're executing the main script.
- Let's say that we call a function, like sqrt.
- This pushes a new frame onto the stack, that remembers where we are executing inside the sqrt function.
- When we hit the return statement, this pops the last frame off the stack.
- The frame underneath tells us where we called sqrt from, so control returns there.

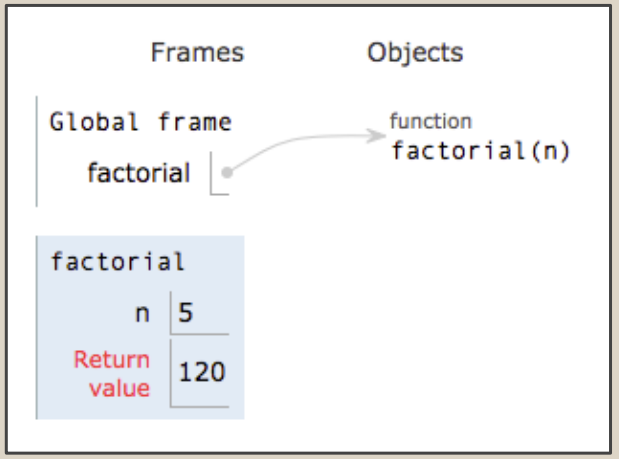




https://docs.python.org/3/library/traceback.html







https://goo.gl/bwpHPo

Let's visit
http://www.pythontutor.com/visualize.html#mode=display to
see the live steps.

Stack Trace | Understanding errors

```
def print_hello(var):
    print("Hello!")
    x = 7 / var
    return x

def some_function(var):
    print("I am the function lord.")
    print(1 + 7 / 3)
    y = print_hello(var)
    print(y)

    return y
```

```
some_function(0)
```

```
I am the function lord.
3.3333333333333335
Hello!

---------------------------------------------------------------
ZeroDivisionError                         Traceback (most r
<ipython-input-11-febbbdbb6e39> in <module>()
----> 1 some_function(0)

<ipython-input-10-3e7fc44e144f> in some_function(var)
      7     print("I am the function lord.")
      8     print(1 + 7 / 3)
----> 9     y = print_hello(var)
     10     print(y)
     11

<ipython-input-10-3e7fc44e144f> in print_hello(var)
      1 def print_hello(var):
      2     print("Hello!")
----> 3     x = 7 / var
      4     return x
      5

ZeroDivisionError: division by zero
```