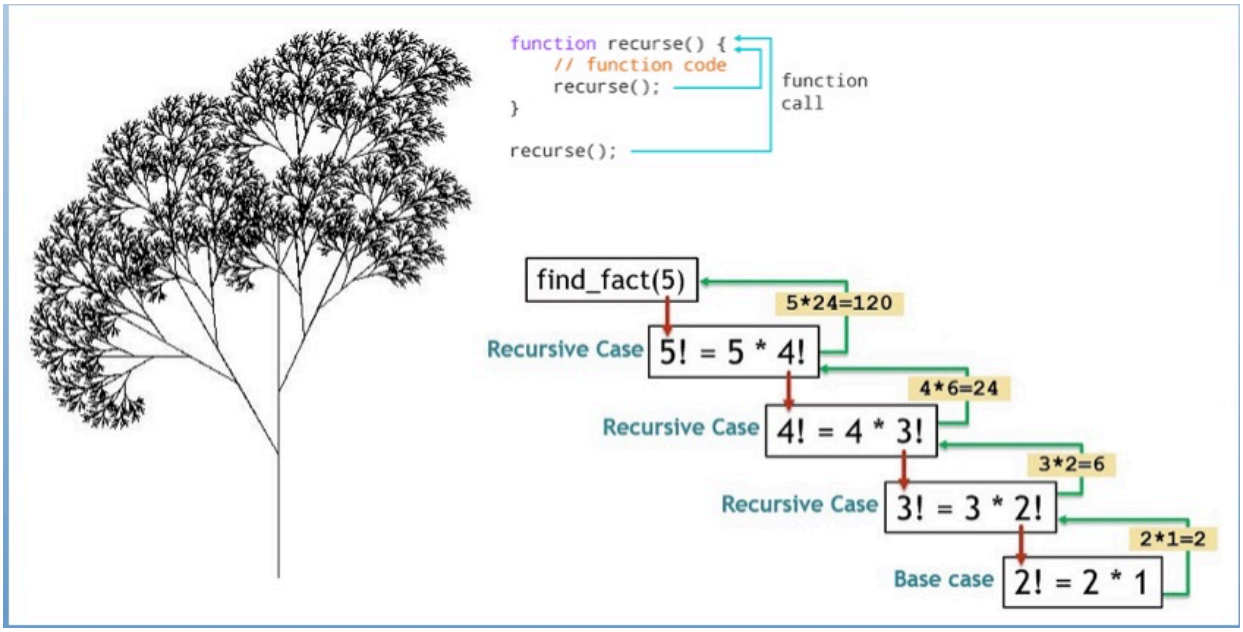
In [27]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= "images/startImage.jpg", width=800, unconfined=True)
```

Out[27]:




# Week 5: Functions, Map, Lambda, SysArg, Recursion, Try/Except

Candidate Week 5, Spring 2022, Jan 31

Today's Topics: Functions and More

- Functions
  - Baseline example
  - Parameters and Default Parameters
  - Errors and Functions
  - Default Values
  - Optional: *arg and **kwarg
  - Keyword Arguments
  - Global and Local Vars
  - Wraping functions
- System Arguments

This notebook has additional demos and explanations that might be useful.

---

## 1. Functions

Overview: Python has a number of function types and features:

- user-defined
- built-in
- lambda (and using map())
- recursive
- a function can call another function (nested)
- functions can have default parameters

and we can use the `type()` command on a function, too, to see that it's an object.

## 1a. Creating a Function: baseline

1. define a function with the keyword def
2. pass zero or more parameters to that function; the parameters are used in the function with scope limited only to that function
3. functions may look outside themselves if you refer to a variable that is not inside the function; for instance, if there's a global variable
4. functions may (but don't have to) return a new value
5. functions are defined - but not used until called in the code
6. functions can take default values, too. E.g., def myFunction(x = 5):

(how does passing a default affect your code?)

## 1b. Demo Functions; parameters and default parameters

In [28]:
```
# ---- FUNCTIONS TO SHOW DEFINITION, 0+ parameters ---
# stripped down function, using the def keyword

def show_the_weather():
    print("The weather today is great!")

# call the function whenever we want and as often as we want
show_the_weather()


def show_the_weather(day, temp):
    # this function accepts 2 parameters
    print(f"{day}'s weather will be {temp} degrees.")

show_the_weather("Thursday", 75)
```

```
The weather today is great!
Thursday's weather will be 75 degrees.
```

In [29]:
```
cat()
cat(i)
cat(i, string)

cat()
cat(5)
cat(5, "hello, kitty")
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-29-8beecf7fe4db> in <module>
----> 1 cat()
      2 cat(i)
      3 cat(i, string)
      4
      5 cat()

NameError: name 'cat' is not defined
```

In [22]:
```
def show_the_weather(day, temp):
     # this function accepts 2 parameters
    print(f"{day}'s weather:")
    print( day_type(temp) )  #this function in another function


def day_type(temp):
    if temp >= 85:
        return "hot!"
    elif temp >= 75 and temp < 85:
        return "pretty warm"
    elif temp >= 50 and temp < 75:
        return "temperate or cool"
    else:
        return "may be cold."

# notice we can use the print() command to accept 0, 1 or more argumen
ts
print("A single parameter in the print method:\n")

print("Today's weather", day_type(65))
print(f"Today's weather is {day_type(65)}")
```

```
A single parameter in the print method:

Today's weather temperate or cool
Today's weather is temperate or cool
```

In [20]:
```
x = 22

def demo_zero():
    print('hurray')

def demo_one(x):
    print(x ** 2)

def demo_two(x, y):
    print(x + y)

def demo_three(x, y, name):
    print(name, "paid $", x+y)

demo_one(5)
demo_two(6,3)
demo_three(20, 42, "Gauri")

# and a default in the parameter
def inflation(x, y, day = 'Today'):
    z = ("The exchange rate for " + day + " is " + str(x*y/100)  + "%"
)
    return(z)


# try the function with and without a day parameter.
inflation(5, 2, "Feb 1, 2021")
inflation(5, 2)
```

```
25
9
Gauri paid $ 62
```

Out[20]:
```
'The exchange rate for Today is 0.1%'
```

---

# 1c. Errors and Functions

In [3]:
```
def myfunction(name):
    print("I accept a single parameter - here, called name:", name)
    print("but what if we don't pass any data?")

myfunction("Dave")
myfunction()
```

```
I accept a single parameter - here, called name: Dave
but what if we don't pass any data?

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-3-90bfb3b948b2> in <module>
      4
      5 myfunction("Dave")
----> 6 myfunction()

TypeError: myfunction() missing 1 required positional argument: 'nam
e'
```

---

## 1d. Default Values and parameters

Note that default names follow last in the list.

In [23]:
```
def myfunction(name = "Tom"):
    print("Welcome,", name)

myfunction("Rex")
myfunction()
```

```
Welcome, Rex
Welcome, Tom
```

## 1e. Optional: *arg and **kwarg

- `*args` is used to unpack a list. "args" is just a name - call it anything you want. The * is used to pass "positional arguments" - that is, not a copy of the full list, but a way to iterate over the positions (0, 1, 2, ... n) to get each element.
- `**kwarg` is for unpacking dictionaries.
</ul> </blockquote>

In [6]:
```
# *args is an unpacking operator.

l1 = [1, 2, 3]
#print(l1)

def sumset(*args):  # NAME doesn't matter - use *cats if you want!
    summer = 0
    for x in args:
        summer += x
        print("x =",x, " ", summer)
    return summer

print("\nSumset with list: ",sumset(*l1))

print("\nSumset with hardcoded list:", sumset(1, 2, 3))
# will work fine.

#but not using the * to indicate unpacking will fail.
print(sumset(l1)) # will fail.
```

```
x = 1    1
x = 2    3
x = 3    6

Sumset with list:  6
x = 1    1
x = 2    3
x = 3    6

Sumset with hardcoded list: 6

---------------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-6-ec346665e50d> in <module>
     17
     18 #but not using the * to indicate unpacking will fail.
---> 19 print(sumset(l1)) # will fail.

<ipython-input-6-ec346665e50d> in sumset(*args)
      7     summer = 0
      8     for x in args:
----> 9         summer += x
     10         print("x =",x, " ", summer)
     11     return summer

TypeError: unsupported operand type(s) for +=: 'int' and 'list'
```

In [7]:
```
# Passing a list of hard-coded names and selecting
# only #2 in unknown list.
guests = ["Tom", "Rex", "Suky", "Lars"]
print(type(guests))

def invitations(*guests):
    print("Who has arrived? ", guests[2])

invitations("Tom", "Rex", "Suky", "Lars")
```

```
<class 'list'>
Who has arrived?  Suky
```

## 1f. Keyword Arguments

Keyword arguments **kwargs are usually associated with dictionaries.

If you don't know how many arguments will be passed, use **

In [41]:
```
def invitations(**people):
  print("Today's host is " + people["host"])

invitations(dj = "Tom", host = "Rex")
```

```
Today's host is Rex
```

With **dictionaries**, you can get data by key or by value:

In [42]:
```
dtest = {"1":"cat", "2":"dog"}
print(type(dtest))

print("\nGETTING VALUES")

def dtester(**kwargs):
    v = ""
    for i in kwargs.values():
        v += i
    return v

print(dtester(**dtest))
```

```
<class 'dict'>

GETTING VALUES
catdog
```

In [43]:
```
print("GETTING KEYS")
dtest = {"1":"cat", "2":"dog"}
print(type(dtest))

def dtester(**kwargs):
    v = ""
    for i in kwargs:
        v += i
    return v

print(dtester(**dtest))
```

```
GETTING KEYS
<class 'dict'>
12
```

---

## 1g. Global and Local Variables

Notice the first instance of "x", "y", and "a" are not in a function - they're unbound and so is visible to all parts of the script.

But the x defined in demo1() is limited in scope - visible only to that function.

In [ ]:
```
x = 5
a = "fish"

print("\nAnd functions ... ")
def demo1():
    x = 27   # demo1.x ≠ global x
    return x

print("Original x: ", x)
print("\t Calling demo1")
print("\t x is ", demo1())

y = 100 # global
def demo2():
    x = -1 # local x
    print("\nCalling demo2():\n\tlocal x * global y =", end="")
    print(x * y)

demo2()

print("_"*40)
```

## 1h. Calling a global var from inside a function. And optional Wrapping and Nesting Functions

Why wrap a function in another function? Something called *information hiding*, among other reasons. Check out Wikipedia (https://en.wikipedia.org/wiki/Nested_function).

Main point: compare the values of global and local (defined in a function) variables.

Optional is the nested demo5().

In [ ]:
```
x = 5
y = 32

print("Global values for x and y: x =",x," id [",id(x),"]  y =",y,"id
[",id(y),"]")

def demo3():
    # usually only local here ... so to reference global ...
    global x
    print("Calling Demo3: Accessing global x: ",x, "and id: ", id(x))

def demo4():
    x = 30  # entirely local!
    y = "dog"
    print("\nCalling Demo4: local x = ",x, " local y = ", y)
    print("\n\tWhat about global x and y? ")

    def demo5():
        global x
        x = -5
        nonlocal y
        y = "cat"

    print("\n","-"*30,"\nInside demo4:\\t Local value of y:", y," and
id:", id(y))
    print("\n\tAbout to call demo5() to update global x and nonlocal y
with nested function...")
    demo5()
    print("\n\t> Local y is 'dog' but nonlocal y is '",y,"' and id:",
id(y))

print("-"*30,"\nStart the demo.\n\n")
print(f"Original values of (global) x {x} and (global) y {y}\n")

demo3()
demo4()

print("-"*30,"\nAfter running demo3() and demo4(), what's the global v
alue of y?")
print("Global y is ", y, " id: [", id(y),"]")

demo5()
```

## 2. System Arguments `sys.argv[]`

Just as we can pass data into a function, so we can pass data from the terminal into our scripts, using `sys.argv`

Executing a .py file from the terminal window means we can provide data to the script from outside that script. The syntax to start a python program also accepts data that are read once the program starts. These are called the argument vector; in code they're a Python list (called argv) from the system module (sys). Hence sys.argv.

What is the value of sys.argv[0]?

In [ ]:
```
# DON'T RUN IN THE NOTEBOOK _ use the argdemo.py from terminal window.
import sys

print("Name of the script", sys.argv[0])

print("# of args: ", len(sys.argv))
print("arguments: ", str(sys.argv))

# cat is choice
for i in range(0, len(sys.argv)):
    print("\t ",i," is ", sys.argv[i])
    cat = sys.argv[3]

print(cat)
```

In [8]:
```
Image(url= "images/argv.png")
# <img src="imags/argv.png"/>
```

Out[8]:
```
gb@gbs-MacBook-Air week_05_alt % python3 argdemo.py 1 2 3 "Dogs"
Name of the script argdemo.py
# of args:  5
arguments:  ['argdemo.py', '1', '2', '3', 'Dogs']
         0  is  argdemo.py
         1  is  1
         2  is  2
         3  is  3
         4  is  Dogs
```

## 3. Try...Except Block

# 3a. Try/Except Basics

Here we look at try/except blocks and using built-in specific and generic exceptions. Later we'll write our own.

Two of the errors are "thrown" when the value of x aren't right - if the end-user enters a non-number value and if there's the chance of division by 0. During debugging you can capture the important errors and catch them and respond as you wish. The except block is for everything else that could happen ...

Unique to python is the additional `else` block: useful for keeping the end-user informed of progress.

There are built-in exceptions and we can define our own exceptions.

In [13]:
```
y = float(input("Enter a float"))
print("reciprocal is ",1/y)
# ------------------------------------------------------------------
---------------

try:
    x = float(input("Enter a number: "))
    print("\tThe reciprocal is ", 1/x)

except ValueError:
    # in case the user enters a letter
    # this is an example of a _specific_ error
    print("Sorry, only numbers, please.")

except ZeroDivisionError:
    # this is an example of a _specific_ error
    print("Sorry, no reciprocal for 0.")

except:
    # this is an example of a _general_ error
    # something else went wrong.
    print("Sorry, there was an error.  Contact Rob.")

else:
    print("\nThis is the else section - called no matter what.")
```

```
Enter a number: 8
        The reciprocal is  0.125

This is the else section - called no matter what.
```

## 3b. Writing our own exceptions

Why would we want to do this?

Noticee the `Exception` class. We're casting the error into a String variable called `e` so we humans can read it. There are different types of Exceptions, too, based on libraries, for instance when using SQL, SQLException (https://dev.mysql.com/doc/connector-python/en/connector-python-api-errors-error.html) (Example below).

In [14]:
```
inventory = ["cheese","bananas","apples"]

try:
    s = input("What do you want? ").lower()
    # reciprocal 1/x -> division by 0 error
    sell(s, 1, inventory)
except Exception as e:
    print("Error: " + str(e))


def sell(item, quantity, inventory):
    print(f"Yes, we have lots of {item}!")

    if item not in inventory:
        raise Exception("Sorry, " + item + " isn't in stock.")
```

```
What do you want? prunes
Error: name 'sell' is not defined
```

---

Example of a specific kind of error, generated when we cannot connect our python script to an SQL server.

```
import mysql.connector


try:
  cnx = mysql.connector.connect(user='scott', database='employees')
  cursor = cnx.cursor()
  cursor.execute("SELECT * FORM employees")   # Syntax error in query
  cnx.close()
except mysql.connector.Error as err:
  print("Something went wrong: {}".format(err))</pre>
```

# 4. Lambda & Map Functions

## 4a. Lambda Functions

The syntax is straight forward: a placeholder variable separated by a colon : and then the action.

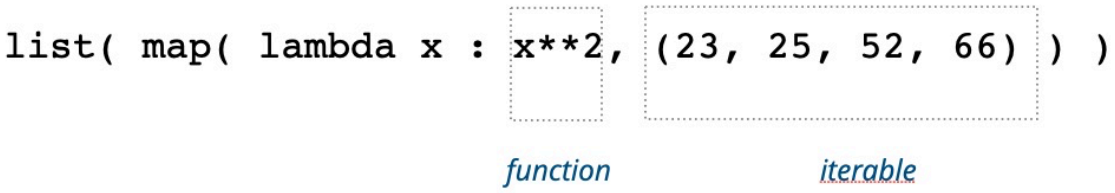
A `lambda` is a small, anonymous function, taking any number of arguments but can have only one expression. The syntax is `lambda` *`arguments`* `:` *`expression`* .

Notice the "placeholder" variable in the middle of the statement, the actions upon (expression) upon that variable. The lambda acts like the "eval()" function in some languages.

Read each of the below statements and think about how they're read by the computer and where the "temp" variable is declared and initialized.

In [30]:
```
Image(url= "images/mapsyntax.jpg")
```

Out[30]:




In the first pass, the value of `x` is taken from the iterable (here 23) and passed to the middle x.

The lambda says to declare a placeholder variable x.

Repeat until the iterable is exhausted.

In [5]:
```
# one var
x = lambda a : a + 10
print(x)
print(x(5))

# two vars:
total = lambda a, b : a + b
print(total(20,15))

# lambda accepting 3 vars and adding them
x = lambda a, b, c : a + b + c
print(x(10, 25, 33))

# notice here number1[0] is added to number2[0] and so on
number1 = [5,10,15]
number2 = [6, 12, 18]

print(type(number2))

result = map(lambda x, y: x + y, number1, number2)

print(list(result))
```

```
<function <lambda> at 0x7fb70c017d30>
15
35
68
<class 'list'>
[11, 22, 33]
```

Nest a lambda into another function ...

In [21]:
```
def get_rate(currency):
    rates = {"EUR": .85, "CND": 1.23}
    return rates[currency]

def todays_rates(n):
  return lambda a : a * n

my_euros = todays_rates( get_rate("EUR") )

money_to_convert = int(input("Enter the amount: $"))
print(f"€{my_euros(money_to_convert)}")
```

```
Enter the amount: $93
€79.05
```

## 3b. Map and Lambda

`map()` executes a specified function for each element in an iterable.

The syntax is `map(`*`function, iterable`*`)`.)

Here we define a function that accepts a single parameter and returns the length of that parameter. The next example passes two iterables. Think of `map()` as your own hands - getting the first value from the iterable and passing it into the placeholder variable x that lambda will use.

In [96]:
```
list(map(lambda x : x **2, (5, 10, 15, 20)))
```

Out[96]:
```
[25, 100, 225, 400]
```

In [10]:
```
# the map command takes each element from the list () and passes
# them one-by-one to the function (whoIsLonger()).  The results
# are still in RAM, so let's send them to a list for printing.

def whoIsLonger(n):
    return len(n)

x = map(whoIsLonger, ('Boston','Berkeley','Roma', 'Juan-les-Pins'))
print(list(x))
```

```
[6, 8, 4, 13]
```

In [11]:
```
# do the same thing different ways
# MAP
def addition(n):
    return n + n

numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(result)
print(list(result))

# try with strings
def additionS(s1, s2):
    return s1 + s2

x = map(additionS, ["Boston", "BackBay", "Cambridge"], ["3000","3000",
"1203"])
print(x)  # notice this is a map object - need to cast into something
usable.

converted_x_to_list = list(x)
print(converted_x_to_list)

# LAMBDA with single var input to iterate over

numbers = (9, 8, 7, 6)
result = map(lambda x: x + x, numbers)
print(list(result))


# LAMBDA with more than 1 objects
number1 = [5, 10, 15]
number2 = [6, 12, 18]

result = map(lambda x, y: x + y, number1, number2)
print(list(result))
```

```
<map object at 0x7fc36732ea90>
[2, 4, 6, 8]
<map object at 0x7fc36732eca0>
['Boston3000', 'BackBay3000', 'Cambridge1203']
[18, 16, 14, 12]
[11, 22, 33]
```

## 3c. Optional: Longer lambda example

See also optional notebooks:

1. **Lambdas, iterators, and more** week_05_functions_intro_extra.ipynb
2. **Content Aggregator** Week-05-ApplicationDemo.ipynb]

In [12]:
```
# Python 3 - sorting a dictionary

counties = {
        "Berkeley":"Oakland", "San Francisco":"San José",
        "Sacramento":"Stockton", "San José":"Hayward",
        "Merced":"Stockton", "Los Feliz":"Santa Monica",
        "Bakersfield":"Mojave","Long Beach":"Crenshaw"}

print("-"*40,"\nNOT sorted:")
for town in counties.keys():
        print ("{:15s} / {:15s}".format(town,counties[town]))

print("-"*40,"\nSorted by KEY:")
towns = list(counties.keys())
towns.sort()
for town in towns:
    print ("{:15s} / {:15s}".format(town,counties[town]))

print("-"*40,"\nSorted by VALUE:")
towns = list(counties.keys())
towns.sort(key = lambda x:counties[x])
for town in towns:
    print ("{:15s} / {:15s}".format(town,counties[town]))
```

```
----------------------------------------
NOT sorted:
Berkeley         / Oakland
San Francisco    / San José
Sacramento       / Stockton
San José         / Hayward
Merced           / Stockton
Los Feliz        / Santa Monica
Bakersfield      / Mojave
Long Beach       / Crenshaw
----------------------------------------
Sorted by KEY:
Bakersfield      / Mojave
Berkeley         / Oakland
Long Beach       / Crenshaw
Los Feliz        / Santa Monica
Merced           / Stockton
Sacramento       / Stockton
San Francisco    / San José
San José         / Hayward
----------------------------------------
Sorted by VALUE:
Long Beach       / Crenshaw
San José         / Hayward
Bakersfield      / Mojave
Berkeley         / Oakland
San Francisco    / San José
Los Feliz        / Santa Monica
Sacramento       / Stockton
Merced           / Stockton
```

---

## Another optional example: sorting and dictionaries using lambda

You cannot sort a dict. If you sort a list of strings that are digits, you don't get a numeric sort by default.

Solutions

- Sort a list of keys
- Specifiy a sort lambda (Python 2 comparator, Python 3 key function) for that list

In [16]:
```
stuff = {'9': 36, '10': 26, '8': 25, '6': 2}

print("-"*40,"\nSorting in NUMERIC order by key")
names = list(stuff.keys())
names.sort(key = lambda x: int(x))

print("-"*40,"\nOutput in NUMERIC order ")
for name in names:
    print(name, stuff[name])

# Following code fills in the gaps for missing keys
# that is, we have 6, 8, 9 but NO 7 as a key.  So we
# make one on the fly and provide it a value of 0
print("-"*40,"\nFilling the 'gaps' with default 0 ")
for want in range(int(names[0]),int(names[-1])+1):
    stuff[str(want)] = stuff.get(str(want),0)

print("-"*40,"\nOutput in NUMERIC order ")
for name in range(int(names[0]),int(names[-1])+1):
    print(name,stuff[str(name)])
```

```
----------------------------------------
Sorting in NUMERIC order by key
----------------------------------------
Output in NUMERIC order
6 2
8 25
9 36
10 26
----------------------------------------
Filling the 'gaps' with default 0
----------------------------------------
Output in NUMERIC order
6 2
7 0
8 25
9 36
10 26
```

# Review of Map and Lambda

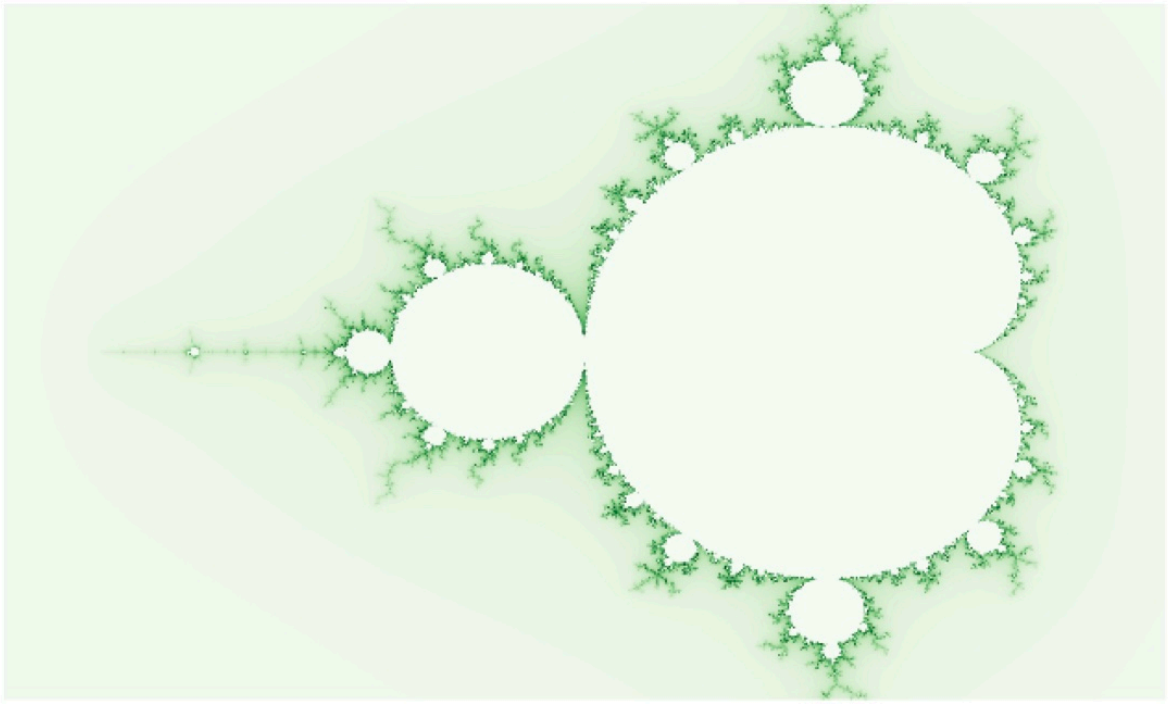
In [31]:
```
# LAMBDA with single var input to iterate over
numbers = (9, 8, 7, 6)
result = map(lambda x: x + x, numbers)
print(list(result))

# OUTPUT: [18, 16, 14, 12]

# LAMBDA with more than 1 object
number1 = [5, 10, 15]
number2 = [6, 12, 18]

result = map(lambda x,y : x+y, number1, number2)
print(list(result))

# OUTPUT: [11, 22, 33]
```

```
[18, 16, 14, 12]
[11, 22, 33]
```

In [ ]:
```
# MAP
def addition(n):
return n + n

numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))


# try with strings
def cityMap(x, y):
   return x + " $" + y

x = map(cityMap, ("Boston", "BackBay", "Cambridge"),
   ("3000","3000","1203"))
print(list(x))
```

In [20]:
```
Image(url= "images/mendelbrot.jpg")
```

Out[20]:



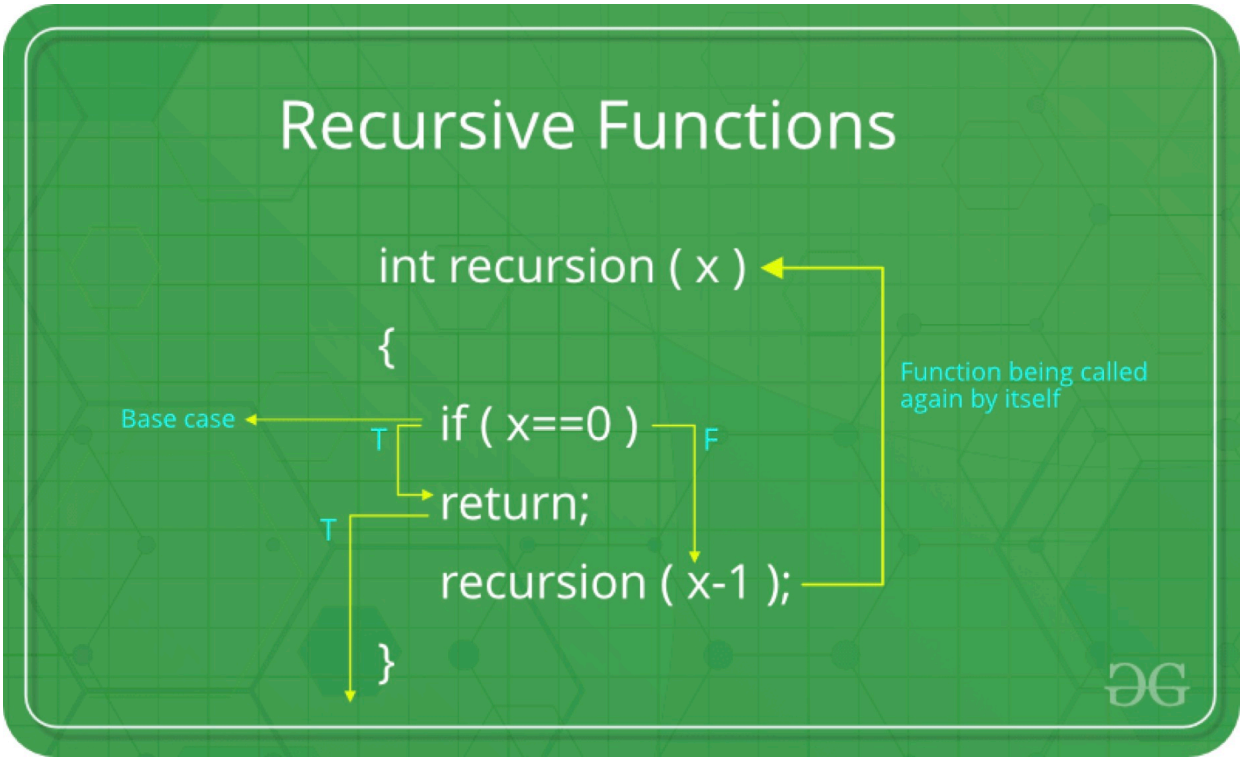
---

## 5. Recursion

When a function calls itself repeatedly, until some condition is met to exit.

In [25]:
```
Image(url= "images/recursion-green.jpg")
```

Out[25]:




Recursion is when a function is declared and in its operation calls itself.

Like all looping behaviors, a recursive call needs an exit condition.
There's "base case" and the "recursive rule", calling itself.

Functions are stored in the stack in RAM; so using 1 function calling itself is better than multiple functions.

Is recursion always a faster technique? (no.)

In [101]:
```
def factorial(n):  # n > 1 ..., say 5 ...
    if n == 1:
        return 1
    return n * factorial(n - 1)

""" notice the function calls itself """
print(factorial(1))
print(factorial(4))
print(factorial(43))
```

```
1
24
604152630633738356373551320685139975072645120000000000
```

In [ ]:
```
def f(i):
    sum = 0

    # are we done?
    counter = 0
    while counter < i:  # so if i = 3 ... 0 < 3
        sum += i

        if counter == i:
            #stop!
        else:
    f(i-1)   #  so if n = 3, then the call is i-1 = 2

# --- RUN IT
f(3)

# CHECK OUT THE OPTIONAL RESOURCES FOR READING AND EXAMPLES.
```

---

## 6. Demonstrate recursion with stack trace

PythonTutor (http://www.pythontutor.com/visualize.html#mode=edit)

In [32]:
```
Image(url= "images/trace.jpg")
```

Out[32]:

```
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
```

# Optional Examples: A little recap and discussion about rounding.

... and then to the breakout rooms!

## Progress of counting data, using return, try/except, raising our own exceptions, etc.</p>

In [22]:
```
def vowel_density(word, vowel="e"):
    """Return fraction of characters within string that match a certai
n vowel."""
    return sum([word[i] == vowel for i in range(0,len(word))])/len(wor
d)
```

In [23]:
```
try:
    word = input("Enter a word: ")
    vowel = input("Enter a vowel: ")
    print("{:.2f} of the letters in {} are {}".format(vowel_density(wo
rd, vowel), word, vowel))
except:
    print("You have entered zero letters so a vowel density cannot be
computed.")
```

```
Enter a word: birmingham
Enter a vowel: u
0.00 of the letters in birmingham are u
```

In [ ]:
```
def vowel_density(word, vowel="e"):
    """Return fraction of characters within string that match a certai
n vowel."""

    if vowel not in "aeiou":
        raise Exception(vowel + " is not a vowel.")

    if not all([word[i] in "abcdefghijklmnopqrstuvwxyz" for i in range
(0,len(word))]):
        raise Exception(word + " is not a valid word.")

    return sum([word[i] == vowel for i in range(0,len(word))])/len(wor
d)
```

In [ ]:
```
def imma_notherfunction(NumLs):
    return list(map(lambda x: x**3, NumLs))

print(imma_notherfunction(range(1,10)))
```

In [ ]:
```
imma_notherfunction = lambda x: [i**3 for i in x]

print(imma_notherfunction(range(1,10)))
```

## Formatting decimals, tests with floats, etc.

There's a lot of rightful confusion about using int, float, decimal, round in computing with python. Python changes the # of decimal places, it seems, depending on what we're doing the variable. The round function is particulary weird this way. When a float is instantiated with many values, they're preserved. compare `x = 4.0` and `y = 4.0000000000` . The command print(x) outputs "4.0". The command print(y) returns "4.0000000000".

In [24]:
```
x = 4
print(x)
print("%.4f" % x)
print(f"{x:.4f}")
```

```
4
4.0000
4.0000
```

## ROUND

it might seem that round would provide what we want but rounding depends on some settings and funkyness of printing versus storing and when performing an arthmetic operation. As the python documentation says "Context precision and rounding only come into play during arithmetic operations." And that python's round has lots of options for rounding up/down, half_even, etc.

Compare these inputs. Here `x = 6.5` . Printing (x) returns 6.5. `print(round(x))` returns 6. [Why not 7?] Check out the `math.ciel()` method.

## DECIMAL AND ROUNDING

The `decimal.Decimal` class is like float - the "significance of a new Decimal is determined solely by the number of digits input." and as noted above "Context precision and rounding only come into play during arithmetic operations."

Check these out: `x = 2.5` print(x) returns just 2.5.
`print(round(5/2)` returns **2** instead of 3.
BUT ... if we convert the variable to a Decimal class and indicate the precision, then the answer is safe(r): `print(round(Decimal(x),2))` returns 2.50

In [13]:
```
f = 6.10392841234
print(f)
print(round(f, 4))
```

```
6.10392841234
6.1039
```

In [9]:
```
import decimal
from decimal import Decimal

x = 1.34
float(x)
round(x, 4)
print(round(x,4))

y = Decimal(3)
print(y)
print(Decimal(float(y)))

x = Decimal(1.34)
type(x)
float(x)
round(x, 4)
print(round(x, 4))
```

```
1.34
3
3
1.3400
```

In [18]:
```
""" with a list of decimals """

declist = (Decimal(4), Decimal(2), Decimal(8), Decimal(3))

print( list(round(c,4) for c in declist) )
```

```
[Decimal('4.0000'), Decimal('2.0000'), Decimal('8.0000'), Decimal('3
.0000')]
```

In [16]:
```
# The print statement will remove the Decimal( ) part of the value.
print(declist)

for i in declist:
    print(round(i,4))
```

```
(Decimal('4'), Decimal('2'), Decimal('8'), Decimal('3'))
4.0000
2.0000
8.0000
3.0000
```

## Decimal with Map/Lambda

In [19]:
```
test = map(lambda x: round(x,4), declist)
print(list(test))

# FLOAT
print("\nFloat test")
test = map(lambda x: float(x), declist)
print(list(test))

print("")
ans = 2 # an int
epi = 0.000000001  # float
print(ans + epi)

print("Compare the outputs:")
# Compare the outputs:
print(ans)
print(round(ans, 2))
print(float(ans))
print(round(float(ans), 4))
print(round(Decimal(ans), 4))
```

```
[Decimal('4.0000'), Decimal('2.0000'), Decimal('8.0000'), Decimal('3
.0000')]

Float test
[4.0, 2.0, 8.0, 3.0]

2.000000001
Compare the outputs:
2
2
2.0
2.0
2.0000
```

Conclusion: there's lots of ways for -format- and to -store- data. Sometimes we may need to cast a var into a different class (like Decimal) in order to round with the output precision we might want.

# Functions & Recursion Extra Notebook Demos

This extra, optional notebook has scripts to demonstrate a variety of activities: working from pseudocode to actual code, looking at functions; creating lambda functions and more.

In the first example, we want to combine skills with lists, parsing, and pseudocoding to create a nice function-based solution to validate characters in a password.

Pseudocode:

valid_password function:

- set the `correct_length` var to `false`
- set the `has_uppercase` var to `false`
- set the `has_lowercase` variable to `false`
- set the `has_digital` var to `false`
- If the password's length is 7 chars or greater:
  - Set the `correct_length` to `true`
  - for each character in the password:
    - if the character is uppercase:
      - set the `has_uppercase` to `true`

    </li> </ul>
    - if the char is a lowercase letter:
      - set the `has_lowercase` to `true`

    </li> </ul>
    - if the character is a digit
      - set the `has_digit` variable to `true`

    </ul> </li> </ul>
    - if `correct_length` and `has_uppercase` and `has_lowercase` and `has_digit` :
      - set the `is_valid` variable to `true`
      - else set the `is_valid` variable to `false`

      </ul> </li>
      - Return the is_valid variable

      </ul>

In [ ]:
```
""" Input: the get_login_name function accepts a first name, a last na
me, and ID as arguments.
    Return: a system login name. """

""" NOTE: if you save this code in a separate file, say login.py,
    you can later <strong>import</strong> it as a module.  See below.
"""

def get_login_name(first, last, idno):
    # get the first 3 letters of the first name.
    # if the name is less than 3 chars, the slide will return entire f
irst name.
    set1 = first[0:3]

    # get the first three of last name ...
    set2 = last[0:3]

    # get ID, if < 3, return entire ID
    set3 = idno[-3:]

    # concatenate
    login_name = set1 + set2 + set3

    # return the login_name
    return login_name

""" The valid_password function accepts a passwor as an argument and r
eturns either true or false.
    The password must have at least 7 letters, at least one upper case
and one lower case, and
    one digit.
"""
def valid_password(password):
    # set the boolean vars to false to get ready ...
    correct_length = False
    has_uppercase = False
    has_lowercase = False
    has_digit = False

    # begin validation test:
    if len(password) >= 7:
        correct_length = True

        """ Test each char and set appropriate flag when required char
is found """
        for ch in password:
            if ch.isupper():
                has_uppercase = True
            if ch.islower():
                has_lowercase = True
```

```
            if ch.isdigit():
                has_digit = True

        """ are all requirements met?  if so, set is_valid to true """
        if correct_length and has_uppercase and \
            has_lowercase and has_digit:
            is_valid = True
        else:
            is_valid = False

        return is_valid
```

Now we want to write a little code to validate the password. Notice the import statement. It's not used here in the notebook, but if you had a separate .py file, say "validate_password.py", you could import the code from the above as a module.

Note line 10 below: if we imported the login module, we would write `while not login.valid_password(password)`

In [ ]:
```
# this program gets the password from the user and validates it.

# import login """ this is a reference to the above snippet if used as
a separate .py file """

def main():
    # get the password from the enduser
    password = input("Enter your password: ")

    # validate - notice we could use while not login.valid_password(pa
ssword)
    while not valid_password(password):
        print("Sorry, that's not a valid password.")
        password = input("Enter your password again: ")

    print(f"-"*40,"Thanks. The password seems legit.")

# call the main function - this is one way.
main()

# I prefer the more pythonic object approach:
# if __name__ == "__main__":
#     main()
```

Here's another example of functions for discussion.
Note the features:

- importing a library
- demo'ing a global variable and local variables
- several functions with different number of arguments
- using the defined functions in the order we wish, controlling 'em in a `main()` function
- calling the functions when we wish by invoking the main()

Review the code. Ignore the display commands - they're just for the notebook.

In [ ]:
```
from IPython.display import Markdown, display
import html


""" Week 05: Functions:
 this script is for in-class discussion of the features and anatomy of
a function
"""

import sys   # a demo for calling tools for the operating system


# demo a global var and local versions
l1 = ["Tom", "Stan", "明娃", "Rex", "लक़समी", "Geraldo"]

# define functions demo'ing different approaches
def about_me():
    display(Markdown((f"\n<text style=color:red>About this script: dem
o of different functions.</text>\n")))

def no_parameters():
    display(Markdown((f"\n<text style=color:red>\nNo Parameters</text>
: \n\tThis is a demo function that accepts no parameters and it does n
ot return anything.")))


def parameter_one(s):
    display(Markdown((f"\n<text style=color:red>\nOne parameter:</text
> \n\tWelcome, to the function, {s}.")))


def parameter_two_with_default(name, lang = 'EN'):
    """ note that the default follows other arguments """
    display(Markdown((f"\n<text style=color:red>\nParameter with two v
```

```
alues - one default</text> (language).\n\tHey, {name}.  Today we'll st
udy {lang}.")))

def find_highest_value(l1, l2):
    """ printing the highest values in two strings, passed as paramete
rs
        NOTE that l1 has global scope and l2 has local scope.
    """

    display(Markdown((f"\n<text style=color:red>Finding the highest va
lue of two lists</text>; lists passed as parameters</text>.\n")))
    print(f"\n\tThe maximum value in this list is: {max(l1)}")
    print(f"\n\tAnd the max for this one of integers is: {max(l2)}")

def return_string(f, l, id):
    """ This function accepts a first name, last name, and an ID - and
returns a password
        takes the first 3 letters of each and jumbles 'em. """
    # for fun will test if id is int or string - we want to use a stri
ng
    id = str(id)
    # the first 3 characters of firstname are returned
    # the 4 chars starting with the second letter and then
    # ID will be less than 3 characters, else the entire ID
    return f[0:3]+l[1:4]+id[-3:]


def all_done():
    display(Markdown((f"\n<text style=color:orange>\n"+"-"*60 + "\nTha
t's it.</text> Notice the use of main() to start the show!\n")))

def main():
    about_me()

    # demos for class
    no_parameters()
    parameter_one("Tom")
    parameter_two_with_default("Fifi")

    # passing objects (two lists)
    l2 = [5,3,1,4,2,6,6,9]
    find_highest_value(l1, l2)

# returning a value: passing data to the call to the function and wrap
ping back the answer
print(f"\nHere's your new easy-to-break password: {return_string('Jack
','Kline',6666)}")

# when starting the script you entered data from the command line, cal
led system argument vector
```

```
# the data are captured from the input in a variable called argv and a
ccessed thru the library sys
# note that we *could* use an if statement to   if len(sys.argv)-1 > 0
:

try:
    print("\nWere there any parameters? ", len(sys.argv)-1)
    print("Here are the arguments from the command line: "+sys.argv[0]
)
    print(f"\nHere's your new easy-to-break password: {return_string(s
ys.argv[1], sys.argv[2], sys.argv[3])}")
except IndexError:
    print("\n* Oh, no! An IndexError was thrown 'cause there aren't en
ough data passed.")

    # close up the script.
    all_done()

# call the main function
main()
```

## Getting ready: using functions and file serialization

This optional snippet is to demo data being saved and shared and the use of dictionaries with our imported data

Saving data, such as dictionaries, to a file should be **serialized**. This means the data are converted to a stream of bytes that are easier to save, retrieve, share. In python, the process of serialization is called **pickling**.

Once you import the pickle module, you'll ...

- open a file for binary writing, using the "wb" argument, for write-binary
- call the pickle module's `dump` method to pickle the object and write it to the selected file
- close the pickle jar (grin).

In [34]:
```
""" demo 1 for pickling """
import pickle

phonebook = {'Chris': "617-555-1212",
             'Tracy': "408-123-4444",
             'Gunnar': "510-333-9292"}

output_file = open("phonebook.dat", "wb")
pickle.dump(phonebook, output_file)
output_file.close()

""" check your harddrive for your phonebook file. """
```

Out[34]:
```
' check your harddrive for your phonebook file. '
```

In [35]:
```
""" demo 2 for pickling object """
import pickle

# main function
def main():
    again = 'y'

    # open a file for binary writing.
    output_file = open('catinfo.dat', 'wb')

    # accept data 'til user wants to stop.
    while again.lower() == "y":
        # get the data and save 'em (yes, "data" are plural; "datum" i
s the singular.)
        save_data(output_file)

        # more data?
        again = input("Enter more data [y/n]: ")

    # close the file.
    output_file.close()

# save_data function gets data and stores them in a dictionary.
# then the dictionary is pickled to the file.
def save_data(file):
    # create an empty dictionary
    cat = {}

    # get the data for the person and store 'em.
    cat['name'] = input("Name: ")
    cat['age']  = int(input("Age: "))
    cat['weight'] = float(input("Weight: "))

    # pickle the dictionary
    pickle.dump(cat, file)

# now start the show by calling the main function
main()
```

```
Name: Fish
Age: 33
Weight: 29
Enter more data [y/n]: n
```

# In this example we have mutible and immutible [list, tuple, dict].

In [ ]:
```
# list - mutable (i.e. changable within itself)
names = ["Pierre","Ryan","Barrette","Cruella"]

# tuple - immutable (i.e. once set up, changes mean rebuilding whole t
hing)
days = ("Monday","Tuesday","Nextday","Thursday")

# dict - mutable and keyed by name
routes = {"X72":"Berkeley - Oakland","272":"San José - Fremont",
        "Zigzag":"Santa Clara - Livermore the pretty way"}

# string can be treated as a list of letters
place = "Triple Rock Brewery"

print(names[1])
print(days[1])
print(place[1])
print(routes["272"])

names[2] = "Graham"; names[3] = "Lisa"
for name in names:
    print("Give some cavier to",name)

# days[3] = "Wednesday"
# Can't assign to tuple member - previous line is a comment
for dau in days:
    print("We will learn on",dau)

# place[0] = "B"
# Can't assign to string member - previous line is a comment
for letter in place:
    print("Gimme a",letter)

routes["C"] = "Boston - Cambridge"  # Adds new - new key
routes["272"] = "Modesto - Santa Barbara"  # Replaces - key exists
for route in routes.keys():
    print(route,"goes between",routes[route])
```

## Looking ahead:

counting data and making subsets by some criteria are important activities. In this example, we import the re (regular expressions) library and use it to parse a file. The test source file could be replaced in real life with data from a live stream, another collection prepared by your team, etc.

In [33]:
```
import os.path
from os import path   # demoing os imports

import re     # for regular expressions
word = re.compile(r'[A-Z]{2,}',re.I)
wordcount = {}

filename = "blog.txt"

""" check if there's a file to process """
if path.exists(filename):

    """ the text is from """
    for line in open(filename):
        words = word.findall(line)
        for item in words:
            i2 = item.lower()
            wordcount[i2] = wordcount.get(i2,0) + 1

    used = list(wordcount.keys())
    try:
        used.sort(lambda y,x:wordcount[x]-wordcount[y])
    except:
        used.sort(key = lambda y:-wordcount[y])

    # 'sample' variable used just for testing / truncating output

    sample = 0
    for item in used:
        print(item + "\t" + str( wordcount[item]))
        sample += 1
        # if sample > 50: break

else:
    print(f"Sorry the file {filename} is not available.")
```

```
Sorry the file blog.txt is not available.
```

## Thinking about functions and planning for OOD.

There's lots of way to run a python script. Consider this OOD-feeling approach of calling main() if there's a python magic method ( `__name__` ) that's been set to `__main__` ...

In [ ]:
```
import os.path

def doesFileExist(fileToFind):
    returnValue = path.exists(filename)
    return returnValue

def printContents(fileToFind):
    print(f"Here are the contents of {fileToFind}.")

def getFileName():
    return input("Enter the file name: ")

def genericErrorMsg(i):
    if i == 1:
        return "Sorry, the file is not found"
    else:
        return "An exception has been logged.  Cannot go on."

def main():
    userFileName = getFileName()
    lineNo = 1

    if doesFileExist(userFileName):
        printContents(userFileName)
        print(f"-"*40,"\n\tNow trying line by line\n")
        f = open(userFileName, "r")
        for line in f:
            print(f"\nLine: {lineNo} = {line} \n")
            lineNo += 1
    else:
        genericErrorMsg(1)

if __name__ == "__main__":
    main()
```

End of the week 5 main points with optional more advanced and applications.

In [ ]: