Lecture 02: Building programs and abstraction in Python

Today we will consider different ways to construct a program in Python (or many languages), and the different implications, pros/cons of each kind.

Working from concrete to abstract, we will consider the following three approaches.

Procedural programming

```
Step-by-step instructions
Structured conditionals (e.g., if/else), and loops (i.e., for)
```

Functional programming

```
procedures that mimic mathematical functions
outputs follow from inputs with no side effects. Side effects are not always bad!
```

Object-oriented programming

```
collections of data *and* functions to operate on that data.
programs as hierarchies of classes
```

We've already seen examples of all three of these approaches.

Procedural programming is basically the same thing as a "cook book"

Our functions, square, and poly are simple examples of functional style.

Our quick intro to complex numbers have elements of object-oriented style.

In [1]:
```
c = 3.1 + 0.2*1j
print(c)

print(c.real)
print(c.imag)

print(c.conjugate())
```

```
(3.1+0.2j)
3.1
0.2
(3.1-0.2j)
```

A complex number is an ***object*** (e.g., `3.1 + 0.2j`).

It has ***attributes*** (eg `.real`, `.imag`) that are bits of smaller data attached to the large data object.

It has ***methods*** (eg `.conjugate()`) that are functions that are attached to the data and can act on the data.

The difference between the `()` at the end of method and attributes explains a lot about calles and objects.

We will talk about object-oriented approachs next week. Today, we are going to see some difference between procedural and functional style.

Tuples

Before we go any further, there is another common Python data type that we didn't cover last time. Ie, a ***tuple***.

We already know about lists, and that they can contain any kind of data types. E.g.,

```
In [2]: lis = [1,'two',3.0 + 0.*1j]
```

A tuple behaves in a very similar way. You use $(\ )$ rather than $[\ ]$.

```
In [3]: tup = (1,'two',3.0 + 0.*1j)
```

```
In [4]: print(lis)
        print(tup)
```

```
[1, 'two', (3+0j)]
(1, 'two', (3+0j))
```

You can **"get items"** for both:

```
In [5]: lis[0] == tup[0]
```

```
Out[5]: True
```

You can change the entries in a list in the following way:

```
In [6]: lis[0] = 5
        print(lis)
```

```
[5, 'two', (3+0j)]
```

Not with a tuple

In [7]:
```
tup[0] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-aeb61b0651cd> in <module>()
----> 1 tup[0] = 5

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

At tuple is like a list, but we say that it is ***immutable***. Once you define it, you can't alter it. You could totally redefine the variable name. But this isn't the same as altering the tuple.

We will find out that this is very useful sometimes. For example, you don't want to change the index of a tensor to something the tensor can't have. More on this as we go.

There is a small way to cheat if you really want. Like a list, a tuple can contain any simple data type. Including tuples and lists.

In [ ]:
```
tup = (1,(2,3),[4,5,6])
print(tup[0])
print(tup[1])
print(tup[2])
```

Becasue tup[2] is a list, we can change it like a list.

In [ ]:
```
tup[2][0]=17
print(tup[2])
```

We can change individual elements of lists. And this list is in a tuple.

But now look at what we've done:

In [ ]:
```
print(tup)
tup == (1,(2,3),[4,5,6])
```

PROGRAM EXAMPLES:

0: How to get from 1 $\rightarrow$ 100 ? Using only +1, and square operations?

In [ ]:
```
def increment(n):
    return n+1
```

In [ ]:
```
increment(3)
```

In [ ]:
```
def square(n):
    return n*n
```

In [ ]:
```
square(3)
```

Procedural way

In [ ]:
```
def find_sequence(initial,goal):
    """Reports back the shortest sequence of increment and square that gi
ves goal starting initial


    Parameters
    ----------
    initial: int
    goal:    int

    """

    # Make a list of tuples, each containing (string, int)
    candidates = [(str(initial),initial)]

    # Loop over all the integers between your start and your goal.
    for i in range(1,goal-initial+1):

        # An empty list
        new_candidates = []

        # Go through each tuple in candidates,
        # For each tuple,
        # put the string into a variable called 'action'
        # put the integer into a variable called 'result'
        for (action,result) in candidates:

            # a = either of the strings  : ' increment' or ' square'
            # r = either of the functions: increment' or square
            # Remember the functions are in our enviroment (aka namespac
e), so we can use them.
            # If we had more functions to try (eg, cube), we could put th
em in the list here.
            for (a,r) in [(' increment',increment),(' square',square)]:

                # put a new tuple on the back of the new_candidates list
                # The string 'adds' the word of the function to the list
 of words
                # The integer is the current function applied to the prev
ious result.
                new_candidates.append((action+a,r(result)))

                # Show how we are doing so far.
                print(i,': ', new_candidates[-1])

                # Check if the latest is the right answer
                if new_candidates[-1][1] == goal:

                    # report back the asnwer
                    return new_candidates[-1]

        # otherwise the list of candidates becomse where we are now.
        candidates = new_candidates
```

In [ ]:
```
answer = find_sequence(1,100)
```

It works:

In [ ]:
```
((1 + 1 + 1)**2 + 1)**2 == 100
```

Functional way

A different approach

In [ ]:
```
def apply(op_list,arg):
    """Applies a list of functions to a single arguement

    Parameters
    ----------
    op_list: a list of function in the enviroment
    arg:     a variable that each function in op_list understands

    """

    # no function of x = x
    if len(op_list) == 0:
        return arg
    else:
        # apply calls it self! It is RECURSIVE
        # Be careful, this can get crazy.
        # Take the right-most element in the list,
        # evaluate it at the arguement, and drop the used function from the li
st.
        return apply(op_list[1:],op_list[0](arg))
```

The enviroment knows what increment and square are now

In [ ]:
```
increment
```

$<\text{function } \_\_\text{main}\_\_.\text{increment}>$ is Python's way of *binding* something to the name increment.

In [ ]:
```
square
```

We can pass anything in the enviroment to functions. Including functions!

Applying nothing to a number better give back the number.

In [ ]:
```
apply([],9)
```

These should behave like you expect:

In [ ]:
```
apply([increment],9)
```

In [ ]:
```
apply([square],9)
```

In [ ]:
```
apply([square,increment],9) == (9**2) + 1 == 82
```

In [ ]:
```
apply([increment,square],9) == (9+1)**2 == 100
```

Now we just need to build up the list of functions needed to get us to our goal.

To do this one step at a time, we need a function that adds complexity

In [ ]:
```
def add_level(op_list,function_list):
    # Make a new list where each element in function_list is appended to op_li
st
    return [x+[y] for y in function_list for x in op_list]
```

The for loop syntax in the return statement is called a **"comprehension"**. It's useful for cleaning up some complicated expressions.

But how does this function work?

In [ ]:
```
add_level([[increment]],[increment,square])
```

What?

`add_level` works with any lists

In [ ]:
```
add_level(add_level([['a','b']],['c','d','e']),['f','g'])
```

In [ ]:
```
L0 = [ [ increment ] ]
L1 = [ [ increment, increment ],
       [ increment, square    ] ]

add_level(L0,[increment,square]) == L1
```

In [ ]:
```
L2 = [ [ increment, increment, increment ],
       [ increment, square,    increment ],
       [ increment, increment, square    ],
       [ increment, square,    square    ] ]

add_level(L1,[increment,square]) == L2
```

You should be able to see the pattern.

Now we just need to combine the two modules `apply` and `add_level`.

In [ ]:
```
def find_sequence(initial,goal):
    """Reports back the shortest sequence of increment and square that gives g
oal starting initial


    Parameters
    ----------
    initial: int
    goal:    int

    """

    # a list with one element, which is the empty list.
    op_list = [[]]

    # Loop over all the integers between your start and your goal.
    for i in range(1,goal-initial+1):

        # op_list --> [ op_list + increment, op_list + square ]
        op_list = add_level(op_list,[increment,square])

        # loop over the list of function lists in op_list
        for seq in op_list:

            # apply each function list to the starting value
            # and see if you get to the goal.
            # If so stop
            if apply(seq,initial) == goal:
                return seq
```

In [ ]:
```
answer = find_sequence(1,100)
```

In [ ]:
```
print(answer)
answer == [ increment, increment, square, increment, square ]
```

In [ ]:
```
apply(answer,1)
```

In [ ]:
```
apply(answer,4)
```

You can always make **"helper functions"** to clean things up

In [ ]:
```
def translate(answer):
    """Tells you what function are in your list in a more readable format

    Parameters
    ----------
    answer: list of functions that are either increent or square

    """

    # An empty string
    L = ''

    # loop over function in answer
    for f in answer:
        if f == increment:
            L = L + ' increment'
        if f == square:
            L = L + ' square'

    print(L)
```

In [ ]:
```
translate(answer)
```

But it only works for `increment` and `square`.

In [ ]:
```
translate(answer + [apply])
```

There is an even better way to do it:

In [ ]:
```
def translate(answer):
    """Tells you what function are in your list in a more readable format

    Parameters
    ----------
    answer: list of *any* functions

    """

    # An empty string
    L = ''

    # loop over function in answer
    for f in answer:
        L = L + ' ' + f.__name__ # Every function has an attribute called "__n
ame__".

    print(L)
```

In [ ]:
```
translate(answer + [apply, translate])
```

The better way uses **object oriented** techniques that we'll discuss more soon.

1. How to exponentiate?

Python already has built-in exponentiation; i.e., b**n = b^n

In [ ]:
```
3**4 == 3*3*3*3
```

We are only using this particular operation because it has nice mathematical properties that we can leverage in code.

Let's start with the most straighforward (**procedural**) way of doing it:

In [ ]:
```
def simple_exp(a,n):
    if n == 0:
        return 1
    else:
        p = 1
        for i in range(n):
            p *= a # this is the same as saying p = p*a
        return p
```

In-place operations

$p *= a$ means something special in Python. It is shorhand for $p = p * a$. It is very nice shorthand (along with +=, -=, and /=).

In [ ]:
```
p = 2
print(p)
p = p * 3
print(p)
```

In [ ]:
```
p = 2
print(p)
p *= 3
print(p)
```

This also illustrates a very important concept in programming. The equals sign doesn't the same thing as in mathematics.

In mathematics, A = B means A and B are the same thing. They are *equal* 24 hours a day, 7 days a week. Always equal.

In programming, A = B means take the value of B and make A have that same value. It is more something like

$$A \leftarrow B$$

And this is exactly how some books on programming write it when describing the structure of algorithms.

Back to our regularly scheduled program...

`simple_exp` might be good enough. Becuase it works:

In [ ]:
```
print(simple_exp(3,30))
simple_exp(3,30) == 3**30
```

But there are other (**functional**) ways:

In [ ]:
```
def recursive_exp(a,n):
    if n == 0:
        return 1
    else:
        return a*recursive_exp(a,n-1)
```

This also works:

In [ ]:
```
recursive_exp(3,30) == simple_exp(3,30)
```

What is the function doing?

```
call recursive_exp(3,8)
    call recursive_exp(3,7)
        call recursive_exp(3,6)
            call recursive_exp(3,5)
                call recursive_exp(3,4)
                    call recursive_exp(3,3)
                        call recursive_exp(3,2)
                            call recursive_exp(3,1)
                                call recursive_exp(3,0)
                                    return 1
                                return 3
                            return 9
                        return 27
                    retrun 81
                return 243
            return 243
        return 729
    return 2187
return 6561
```

In [ ]:
```
print(3**8)
print(recursive_exp(3,8))
```

Use math if you can.

A really nice feature of recursive progtamming is that it a lot of mathematical structures are built the same way.

$$Fish + Fish + Fish + Fish + Fish + Fish = 6\,Fish.$$

versus

$$5Fish + Fish = 6\,Fish$$

versus

$$2 \times (3\,Fish) = 6\,Fish$$

How does this apply here?

First consider Python the module operator %

In [ ]:
```
for i in range(10):
    print((i,i%2,i%3,i%4,i%5))
```

n%a is the **remainder* for n/a

The module operator goes hand-in-hand with the **floor-divide** operator //

In [ ]:
```
for i in range(10):
    print((i,i//2,i//3,i//4,i//5))
```

n//a just divides n by a and throws aways the non-integer part.

For any positive integers (n,a),

```
a*(n//a) + n % a  = n
```

These opeartors are very useful in many situations.

In [ ]:
```
for i in range(10):
    print((i==2*(i//2)+i%2,i==3*(i//3)+i%3,i==4*(i//4)+i%4,i==5*(i//5)+i%5))
```

Now we can check to see if a number is divisible by 2. And use a mathematical property of exponentiation.

$$a^n = (a * a)^{n/2}, \quad \text{if} \quad n = \text{even}.$$

In [ ]:
```
def fast_exp(a,n):
    if n == 0:
        return 1
    elif n%2==1:
        return a*fast_exp(a,n-1)
    else:
        b = fast_exp(a,n//2)
        return b*b
```

It still works:

In [ ]:
```
print(fast_exp(3,30)==recursive_exp(3,30))
print(fast_exp(3,30)==simple_exp(3,30))
```

```
call fast_exp(3,30)
    call fast_exp(3,15)
        call fast_exp(3,14)
            call fast_exp(3,7)
                call fast_exp(3,6)
                    call fast_exp(3,3)
                        call fast_exp(3,2)
                            call fast_exp(3,1)
                                call fast_exp(3,0)
                                return 1
                            return 3
                        return 9
                    return 27
                retrun 729
            return 2187
        return 4782969
    return 14348907
return 205891132094649
```

This is the same number of recursive call if we had called recursive_exp(3,8) and about 30% the numebr than if we had called recursive_exp(3,30)

For large n, fast_exp(a,n) requires about log(n) the number of calls as recursive_exp(a,n). This can be a huge savings in general.

Divide and Conquer

This is a very simple example of what's known as a ***Divide and Conquer*** algorithm. They show up in a lot of places. Recursive functional programming makes it simple it implement.

This is the idea that a program is *expressive*.

Expressiveness.

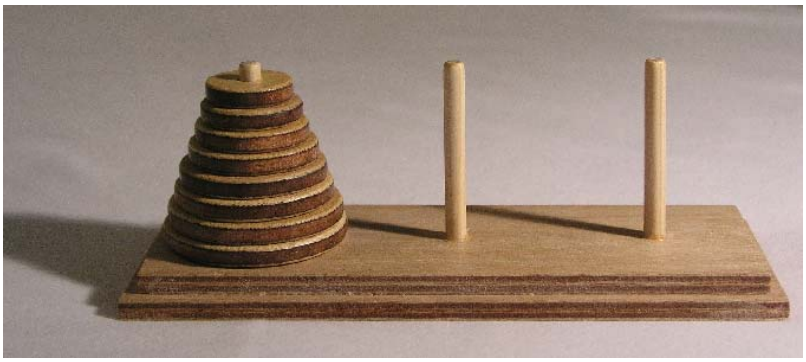
One of the best reasons Python is a good language is that it is ***expressive***. That means that it is easy to *express*, or ***encode*** complicated ideas realitively easily. In `recursive_exp` we used the mathematical structure of the thing we were trying to calculate to save us works. We could express this easily in the form of a recursive function call.

Sometimes that expressiveness is almost spooky...

2: Tower of Hanoi

You can learn a lot about this game, and it's rules on wikipedia:

https://en.wikipedia.org/wiki/Tower_of_Hanoi (https://en.wikipedia.org/wiki/Tower_of_Hanoi)



The point of this problem is that we don't know how to solve it. But we can solve it is we can get one step away from a problem one unit smaller. That is what this recursive function does.

The following program works:

In [ ]:
```
def Hanoi(n,A,B,C):
    if n==1:
        print(A + ' --> ' + B)
    else:
        Hanoi(n-1,A,C,B)
        Hanoi(1,A,B,C)
        Hanoi(n-1,C,B,A)
```

I wrote the above just so you can see how simple something can be: *7 lines*. But to provie some more context, here is the same program fully commented, and with some other bells and whistles:

In [ ]:
```
def Hanoi(n,A,B,C,k=0):
    """solves the Tower of Hanoi probelm.

    The recursive scheme finds the moves to take
    n disks from A to B using C as an auxiliary hold area.

    Parameters
    ----------
    n: int number of disks
    A: string with name of the starting posts.
    B: string with name of the finishing post.
    C: string with name of the auxiliary post.
    k : optional arguement giving the previous number of moves.
    """

    if n==1:
        k += 1 # in-place update k = k + 1
        print(k,":",A + ' --> ' + B)
    else:

        # move n-1 disks from A to C, using B as the auxiliary
        k = Hanoi(n-1,A,C,B,k=k)

        # move 1 disk from A to B using C as the auxiliary.
        k = Hanoi(1,A,B,C,k=k)

        # move n-1 disks from C to B, using A as the auxiliary
        k = Hanoi(n-1,C,B,A,k=k)

    return k # new number of moves.
```

Most people could probably find the following solutions by hand:

In [ ]:
```
k = Hanoi(1,'a','b','c')
```

In [ ]:
```
k = Hanoi(2,'a','b','c')
```

In [ ]:
```
k = Hanoi(3,'a','b','c')
```

Some people might be willing to find these solutions of this by hand.

In [ ]:
```
k = Hanoi(4,'a','b','c')
```

In [ ]:
```
k = Hanoi(5,'a','b','c')
```

I doubt many people would want to try this.

In [ ]:
```
k=Hanoi(10,'a','b','c')
```

Here is an alternative method:

In [ ]:
```
def post(n,i):
    if i==0: return 'a'
    if (i+(n%2))%2==1 : return 'b'
    return 'c'

def Hanoi_moves(n):
    for k in range(1,2**n): print(k, ":", post(n, (k & k - 1) % 3 ) + ' --> '
+ post(n, ((k | k - 1) + 1) % 3 ))
```

In [ ]:
```
Hanoi_moves(4)
```

In [ ]:
```
k = Hanoi(4,'a','b','c')
```

How the Procedural Hanoi program works is not essential. It uses fancy bit-wise operations (i.e., & , |). Understanding it relates to the deeper mathematical properties of the Hanoi game. But the point is that *in this instance* we were able to take a ***recursive*** function and turn it into one that computes a given step without know the previous steps.

```
We can always translate a procedural function into a recursive function.

We cannot always do the reverse.

We cannot know for sure if we can translate a recursive function into a procedural
 without basically finding it.
```

Therefore

$$\text{Recursive} \quad > \quad \text{Procedural.}$$

But we don't knwo how much bigger.

I highly reccomend reading the following overview of the subject of incompleteness in mathematics by Gregory Chaitin

https://arxiv.org/abs/math/0411091 (https://arxiv.org/abs/math/0411091)

I've also posted this in the Extras section on ed.