Key word arguments, line and cell magics, writing your own modules

kwargs - accept an arbitrary number of keyword arguments (not just a fixed number of arguments with unknown size)

In [ ]:
```
def define_user(name, **info):
  """Accept user input for database

  input:
    name - first name of user
    **info - kwargs...other relevant info in form of a dictionary

  returns:
    dictionary with user info
  """

  # init a blank dictionary
  user_info = {}

  user_info['name'] = name

  # then loop over kwargs
  for k, v in info.items():
    user_info[k] = v

  return user_info
```

Can pass in arguments using keywords defined in the function call

In [ ]:
```
usr_info = define_user('john', bike='giant', car='tacoma', house='hearst castle')
usr_info
```

Magics are special functions that are supported by the IPython kernel (the thing that interprets your python code and turns it into something the machine can understand).

- Different kernels have different magics.
- Line magics are called using the % syntax
- Cell magics, that operate on an entire cell worth of code, use the %% syntax
- link to good ref

Some handy line magics

- my favorite is 'timeit'

In [ ]:
```
import random
```

In [ ]:
```
# will work on the entire line of code that you enter...note that i'm sticking
# the entire while... statement in one line here.
%timeit for i in range(0,100):  x = random.random() * 10
```

Can create an alias for a line magic

- usually not such a great idea, but can be handy if you're re-using the same magic over and over again.
- don't sacrifice short names for readability of code!

In [ ]:
```
%alias_magic t timeit
```

In [ ]:
```
%t for i in range(0,100):  x = random.random() * 10
```

Some others...

figure out the current directory (folder) that you are in

In [ ]:
```
%pwd
```

List the contents of the current folder

In [ ]:
```
%ls
```

Change the current directory

In [ ]:
```
# drop down a level in the directory tree
%cd ..
%ls
```

In [ ]:
```
# then go back to the content folder we were just in and list contents
%cd /content/
%ls
```

And 'who' - figure out what variables/functions are in memory

In [ ]:
```
%who
```

And 'whos' gives even more details about type and data. It's generally more useful when you get used to it...

In [ ]:
```
%whos
```

Some handy cell magics

In [ ]:
```
%%timeit
i = 0
x = 0
for y in range(0,5):
  x = i**2 + 10*random.random()
```

Figure out all available line and cell magics on your kernel

In [ ]:
```
%lsmagic
```

modules! (libraries)

Define our own module to carry out some common math functions

- use the writefile magic to write out the def of 4 functions to a .py file.
- That .py file (which is really just a text file) will define the module and you can load it and gain access to the functions

In [ ]:
```
%%writefile math_tools.py

def square(x):
    y = x ** 2
    return y

def cubed(x):
    y = x ** 3
    return(y)

def squrt(x):
    y = x ** (1/2)
    return(y)

def times_ten(x):
    y = x * 10
    return(y)
```

Show the contents of the module...in collab use the %pycat line magic

In [ ]:
```
%pycat math_tools.py
```

In [ ]:
```
%ls
```

Import using 'as' to create an alias

- works just like an object, with the functions called like methods using the object.method type notation
- just import the module once per notebook, once you run it the contents will be part of the global namespace and will be accessible by any code cell in the notebook

In [ ]:
```
import math_tools as mt
```

In [ ]:
```
mt.square(10)
```

Can also import just a specific function so that you can call it directly

In [ ]:
```
from math_tools import cubed
```

In [ ]:
```
cubed(2)
```

Need to be very careful here!

- add a 'list' function to our module...what happens?

In [ ]:
```
%%writefile math_tools.py

def square(x):
    y = x ** 2
    return y

def cubed(x):
    y = x ** 3
    return y

def squrt(x):
    y = x ** (1/2)
    return y

def times_ten(x):
    y = x * 10
    return y

# list out our numbers
def list(x):
  for i in x:
    print(i)
```

You might give something the same name as an important built in function...and in this case, it does something else that will cause errors down the road

In [ ]:
```
from math_tools import list
```

In this case we don't really hurt anything...

In [ ]:
```
list([10,20])
```

But in this case things go totally wrong and you will tear your hair out figuring out what the bug is

In [ ]:
```
x = list(range(0,10))

y = 0
for i in x:
  y += i

print(y)
```

You can combine what we've learning so far to also give a specific function from a module an alias

- again - rule here is that you should not sacrifice readability for short names...

In [ ]:
```
from math_tools import cubed as cb
```

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

Bonus: Recursion...function calling itself!

In [ ]:
```
def factorial(n):
  if n == 1:
    print('end of recursion')
    return 1
  else:
    print('current value', n)
    result = n * factorial(n-1)
    return result
```

In [ ]:
```
print(factorial(4))
```