Try...catch statements, intro to functions

try, except, finally, else

- handle errors gracefully
- try something, if it works then fine, but if not then print a user friendly message explaining what went wrong.

In [ ]:
```
print(y)
```

In [ ]:
```
try:
  print(y)

except:
  print('y is not a thing, why are you trying to print it???')
```

You can also catch specific errors to provide even more specific feedback about what happened.

- same general structure, just with specific messages

In [ ]:
```
try:
  print(y)

# if its a name error (like something not yet defined)
except NameError:
  print('y is not defined')

#  if anything else bad happened, do this...
except:
  print('some error happened, not a name error')
```

Else statement attached to try...will execute if the statements in the "try" block complete without error

In [ ]:
```
# actually define y first so you don't get a name error
y = 10

try:
  print(y)

# if its a name error (like something not yet defined)
except NameError:
  print('y is not defined')

#  if anything else bad happened, do this...
except:
  print('some error happened, not a name error')

# because the try block will execute without error, the statements in this
# else block will execute as well...
else:
  x = y+10
  print(x)
```

Can catch multiple types of errors...

In [ ]:
```
y = '27'
x = '10'

# z = 10
# w = 0

try:
  #print(x * y)
  print(z/w)

except (TypeError):
  print('TypeError')

except (ZeroDivisionError):
  print('ZeroDiv Error')
```

Finally executes regardless of try/except outcome

In [ ]:
```
x = 'john'
try:
  print(z)

except NameError:
  print('y throws a name error')

finally:
  print('you finished the try-except code block')
```

functions...

- write a chunk of reusable code for operations that you will perform repeatedly in the context of a given application.
- always include information about what the function does, and what the input/output parameters are!
  - "docstring"

In [ ]:
```
def print_msg(msg,postfix):
  """a simple little function to print a msg

  inputs:
  msg = what you want to say at the start of msg
  postfix = what you want the msg to end with
  """
  print(msg, postfix)
```

In [ ]:
```
print_msg?
```

In [ ]:
```
print_msg('my name is','john')
```

```
my name is john
```

positional arguments vs keyword arguments

In [ ]:
```
print_msg('john','my name is')
```

In [ ]:
```
print_msg(postfix='john',msg='my name is')
```

return values

- have your function return a value after performing some computation

In [ ]:
```
def add_two(x):
  """ a function to add 2

  input: integer
  output: that integer + 2
  """
  x = x + 2
  return x
```

In [ ]:
```
add_two(10)
```

In [ ]:
```
def add_two(x,y):
  x = x + 2
  y = y + 2
  return x,y
```

In [ ]:
```
### multiple return values.
val = add_two(10,12)
type(val)
val[0]
```

If a variable is defined within a function, then it is in the local namespace - it is created when the function is called and then destroyed after the function returns. Variables created outside of functions are in the global namespace and can be accessed from any cell in the notebook.

In [ ]:
```
# define a variable in the local namespace of the function
def minus_two(x):
  # z is a local variable - does not exist outside of function
  z = x - 2
  return z
```

In [ ]:
```
print(minus_two(2))

try:
  print(z)

except:
  print('error')
```

So notice that things like the sort method will modify the list, even though its "local" to the function

- this is because the list was defined in the **global** namespace, so if you modify it within the function, you will also modify it in the global namespace

In [ ]:
```
def sort_list(in_list):
  in_list.sort()
```

In [ ]:
```
my_list = ['john', 'ella', 'jack']
sort_list(my_list)
print(my_list)

# note - we don't return anything here...the list is modified internally by the
# function
```

But if you used sorted() (or something else that doesn't perm modify the list) then you would need to return the sorted value

In [ ]:
```
def new_sorted_list(in_list):
  sort_list = sorted(in_list)
  return sort_list
```

In [ ]:
```
my_list = ['john', 'ella', 'jack']
s_my_list = new_sorted_list(my_list)
print(my_list)
print(s_my_list)
```

Make a function that allows variable length argument list using the *args special syntax

In [41]:
```
# compute the cumulative product of a list of numbers
def mult(*numbers):
  z = 1 # why initializing to 1???
  for num in numbers:
    z *= num

  return z
```

In [51]:
```
mult(10,20,30,10)
```

Out[51]:
```
60000
```

In [ ]:
```
# compute mean across a list of numbers - and list can be as long as you'd like!
def comp_avg(*nums_to_avg):
  """compute mean across a list of numbers"""
  avg = sum(nums_to_avg)/len(nums_to_avg)
  return avg
```

two ways to call...pass in vals directly

In [ ]:
```
comp_avg(10,20,30,40,50)
```

or pass in a list...use the *

In [ ]:
```
nums = [10,20,30,40,50]
comp_avg(*nums)
```

Bonus stuff about using while loops...

In [ ]:
```
# fill up a dictionary with key, value pairs.

# initialize the dictionary
fav_food = {}

while True:
  name = input('What is your name: ')
  food = input('What is your fav food: ')

  fav_food[name] = food

  more = input('Enter any more name/food pairs[y/n]? ')
  if more == 'n':
    break

for name, food in fav_food.items():
  print(name, 'likes', food)
```

In [ ]:
```
# more relevant example for experiments
stim_trigger = ''

while True:
  stim_trigger = input('Start stimus presentation[y/n] ')

  if stim_trigger == 'y':
    print('starting experiment')
    break
  elif stim_trigger == 'n':
    print('what? you want to start...')
  else:
    print('answer only y or n')

print('this is where the stimulus presentation code would go')
```