

SELECTED TOPICS IN ENGINEERING

*INTR. TO PROG. FOR DATA SCIENCE*  
ENGR 350

Tuesday–Thursday 10:00–12:45

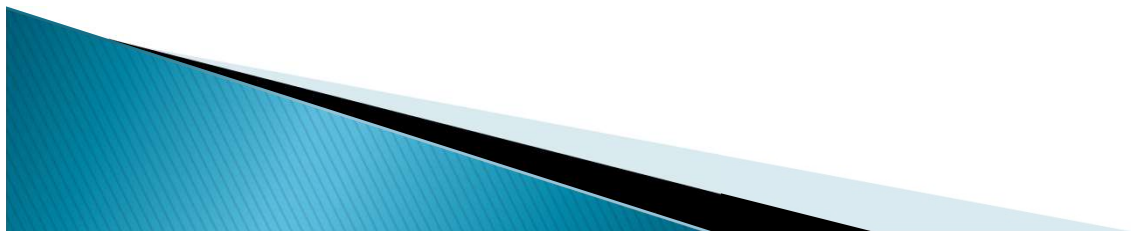
ENG B05

2019 Summer

Dr. Banu Yobaş

# Modularization

- ▶ Breaking down a large main program into modules
- ▶ Statements taken out of a main program and put into a module have been encapsulated
- ▶ Main program shorter and easier to understand
- ▶ Reasons
  - Abstraction
  - Allows multiple programmers to work on a problem
  - Reuse your work more easily

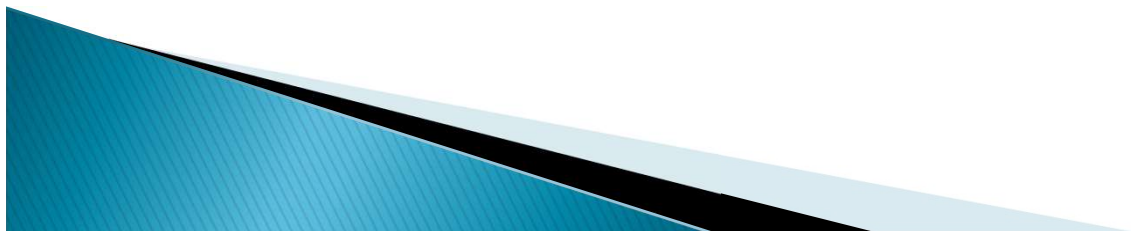


# Reusability & Reliability

## Reusability

- ▶ Feature of modular programs
- ▶ Allows individual modules to be used in a variety of applications
- ▶ Many real-world examples of reusability

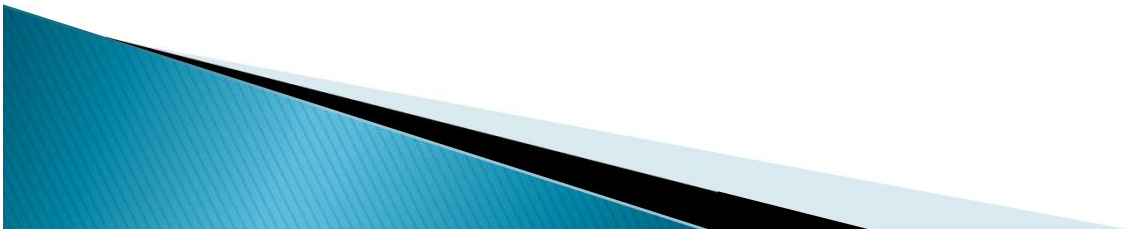
**Reliability :** Assures that a module has been tested and proven to function correctly



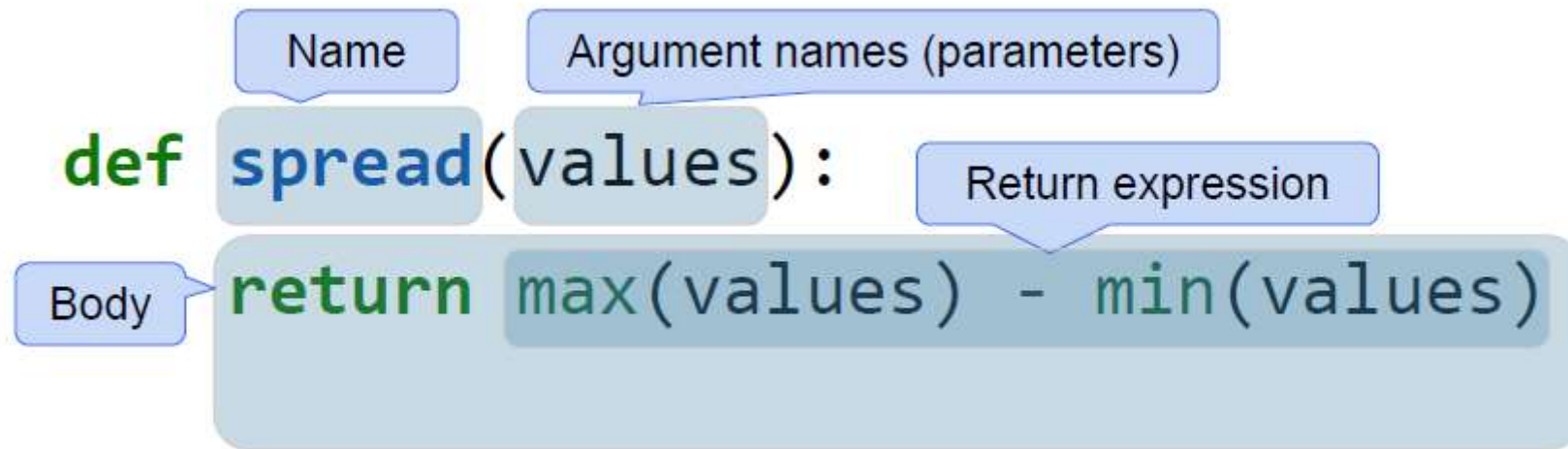
# Functions, Procedures

```
def name(arg1, arg2, ...):  
    """documentation"""           # optional doc  
    string  
    statements  
  
    return                          # from procedure  
    return expression             # from function
```

Pass



# Example Function



```
def gcd(a, b):  
    "greatest common divisor"  
    while a != 0:  
        a, b = b%a, a    # parallel assignment  
    return b
```

```
>>> gcd.__doc__  
'greatest common divisor'  
>>> gcd(12, 20)  
4
```

- ▶ What does this function do? What kind of input does it take? What output will it give? What's a reasonable name?

```
def f(s):  
    return np.round(s / sum(s) * 100, 2)
```



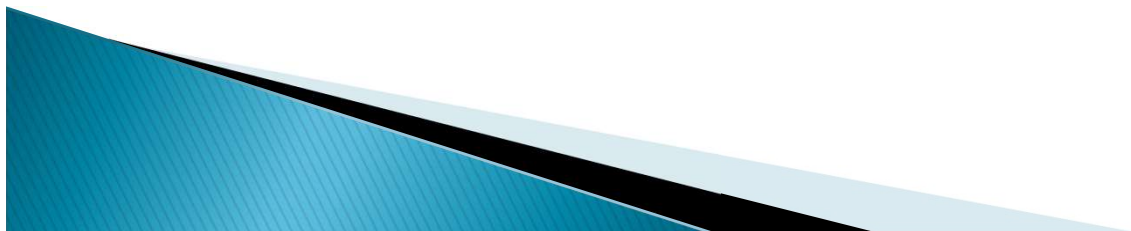
# docstring – “documentation”

- ▶ A docstring is a string at the beginning of a function that explains the interface.



# Parameters and arguments

- ▶ Inside the function, the arguments are assigned to variables called parameters
- ▶ When you create a variable inside a function, it is local
- ▶ Parameters are also local
- ▶ you should avoid having a variable and a function with the same name.





# Function return values

- ▶ Void functions might display something on the screen or have some other effect, but they don't have a return value.
- ▶ If you assign the result to a variable, you get a special value called None

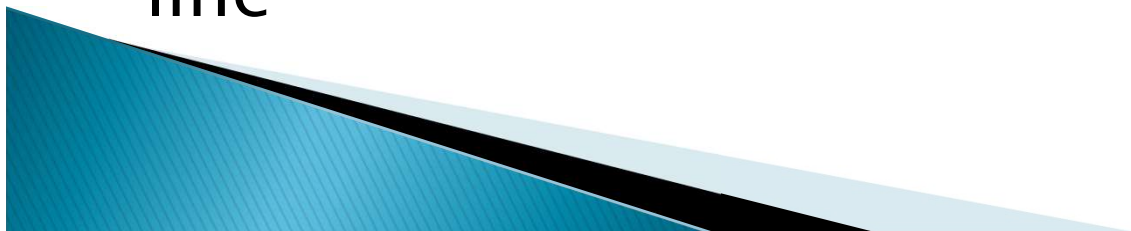
```
result= a_void_func(a,b)
```

```
print(result)
```

```
line=['a', 'x' 'dag' 'cb' 'bac']
```

```
print(line.sort())
```

```
line
```



# Function()

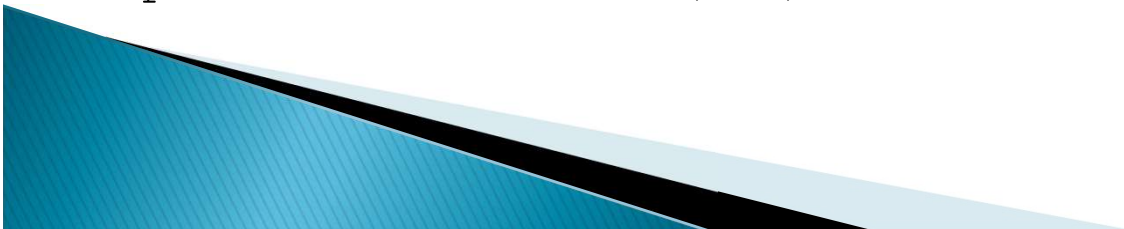
- ▶ Sometimes a function needs to return more than one value. This can be done by simply listing more than one expression in the return statement.

- ▶ As a silly example

```
def sumDiff(x, y):  
    sum = x + y  
    diff = x - y  
    return sum, diff
```

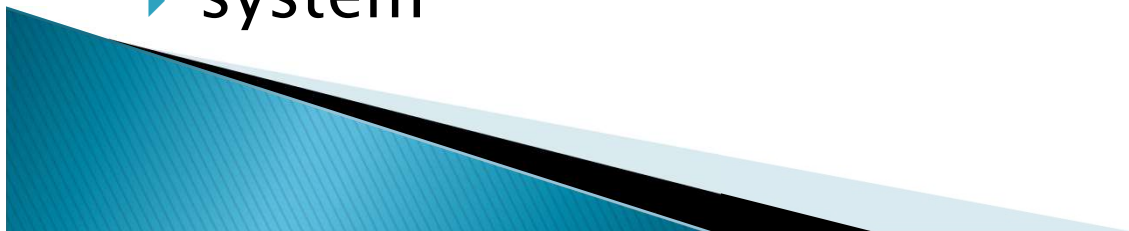
- ▶ This return hands back two values. When calling this function, we would place it in a simultaneous assignment.

```
num1, num2 = input("Please enter two numbers (num1,  
    num2) ")  
s, d = sumDiff(num1, num2)  
print "The sum is", s, "and the difference is", d
```



# Modules

- ▶ Functions that operate on a particular type of object  
`object.method(maybe some arguments)`
- ▶ String methods operate on strings
  - `'APPLE'.lower()` has the value `'apple'`
  - `'APPLE'.replace('A', 'SNA')` has the value `'SNAPPLE'`
- ▶ `math`
- ▶ `string`
- ▶ `random`
- ▶ `os`
- ▶ `system`



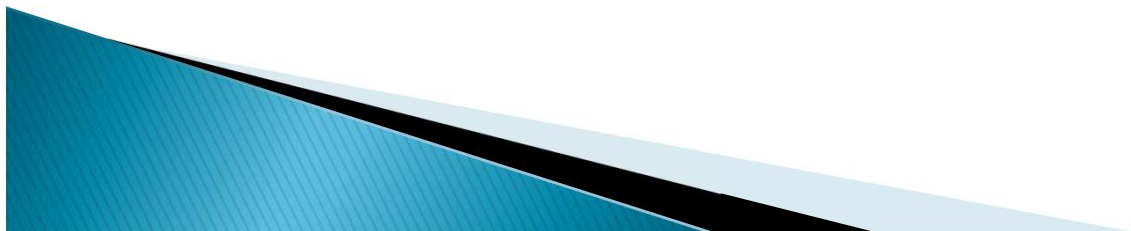
# map()

- ▶ Takes a list and applies a given function to all elements of the list, returning a list.

`map(func_name, list)`

OR

- ▶ expects a function object and any number of iterables like list, dictionary, etc. It executes the function\_object for each element in the sequence and returns a list of the elements modified by the function object.

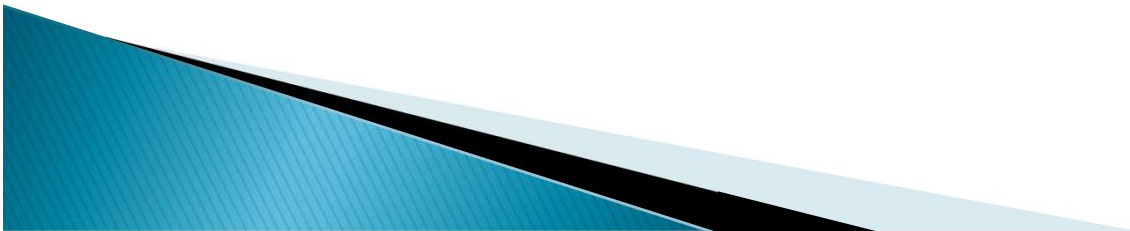


# map()

We may wish to transform all elements in a list. For example, we may wish to turn all words in a file to lower case

The general flavor of this is the following.

```
def map(file):  
    lst = []  
    for word in file:  
        lower_case = word.lower()  
        lst.append(lower_case)  
    return lst
```



# reduce()

- ▶ An operation that combines a sequence of elements into a single value is sometimes called **reduce**.
- ▶ Reduce arises when we wish to summarize.
- ▶ Sum of the numbers in a list [3,4,3,5]
- ▶ The Word Count problem: how many characters are in this file?

The general flavor of this is the following, for some function count that tells what this item is worth

```
def reduce(file):  
    sum = 0  
    for word in file:  
        increment = count(word)  
        sum = sum + increment  
  
    return increment
```



# random module

```
import random
random.choice(['apple', 'pear', 'banana'])
'apple'
random.sample(range(100), 10) # sampling without
replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
random.random() # random float
0.17970987693706186
random.randrange(6) # random integer chosen from range(6)
4
random.seed()
```



# random module

- ▶ Python uses the Mersenne Twister as the core generator.
- ▶ It produces 53-bit precision floats and has a period of  $2^{19937}-1$ .
- ▶ The underlying implementation in C is both fast and threadsafe.
- ▶ The Mersenne Twister is one of the most extensively tested random number generators in existence.
- ▶ However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.





# random.seed()

- ▶ If we do not give a seed, the random module sets a seed based on the current time.
- ▶ The seed will be different each time we run the program and consequently the sequence of random numbers will also be different from run to run.

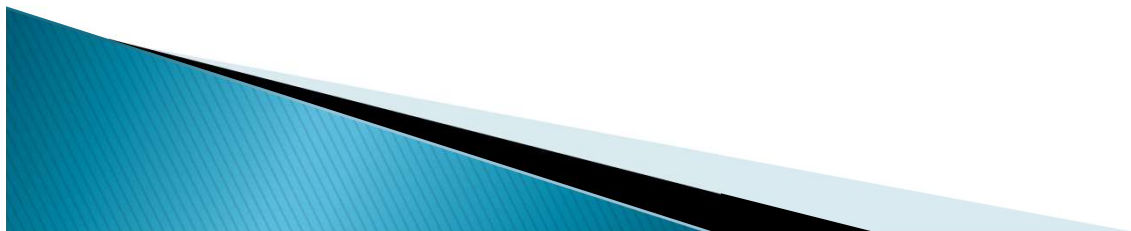


How many heads in 100 tosses?



# Multiplication Rule

- ▶ Chance that two events  $A$  and  $B$  both happen  
=  $P(A \text{ happens}) \times P(B \text{ happens given that } A \text{ has happened})$
- ▶ The answer is *smaller* than each of the two chances being multiplied.
- ▶ The more conditions you have to satisfy, the less likely you are to satisfy them all.

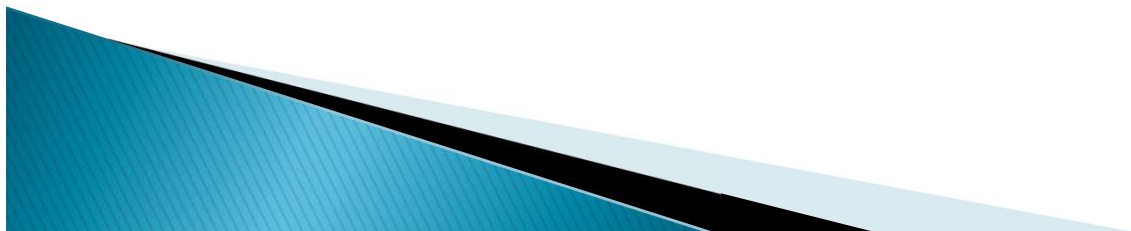


# Addition Rule

- ▶ If event  $A$  can happen in *exactly one* of two ways, then

$$P(A) = P(\text{first way}) + P(\text{second way})$$

- ▶ The answer is *bigger* than each the chance of each individual way.



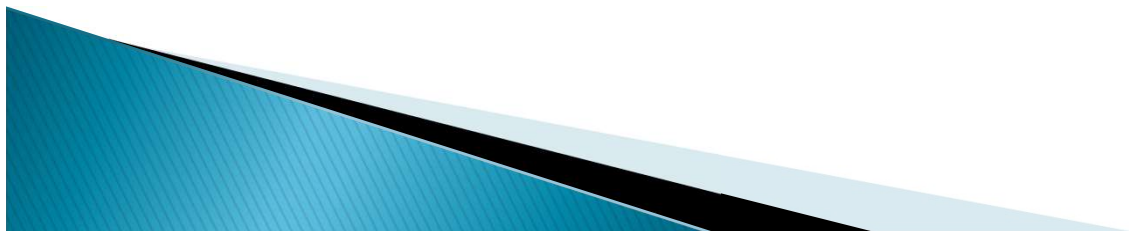
# At Least one Head

## ► In 3 tosses:

- Any outcome *except* TTT
- $P(\text{TTT}) = (1/2) \times (1/2) \times (1/2) = 1/8$
- $P(\text{at least one head}) = 1 - P(\text{TTT}) = 7/8 = 87.5\%$

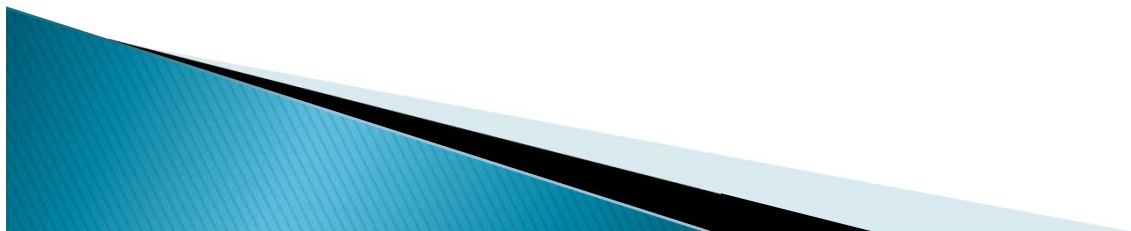
## ► In 10 tosses:

- $1 - (1/2)^{10}$
- 99.9%



# Sampling

- ▶ **Deterministic sample:**
  - Sampling scheme doesn't involve chance
- ▶ **Probability sample:**
  - Before the sample is drawn, you have to know the selection probability of every group of people in the population.
  - Not all individuals have to have equal chance of being selected.



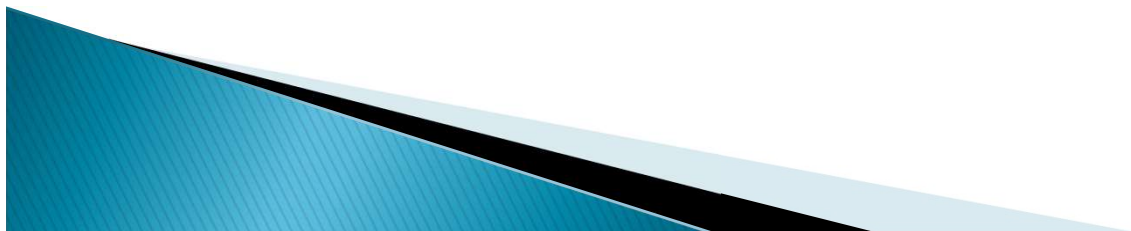
# Sample of Convenience

- ▶ Example: sample consists of whoever walks by
    - Just because you think you're sampling "at random", doesn't mean you are.
  - ▶ If you can't figure out ahead of time
    - what's the population
    - what's the chance of selection, for each group in the population
- then you don't have a random sample.



# Law of Averages

- ▶ If a chance experiment is repeated many times, independently and under the same conditions, then the proportion of times that an event occurs gets closer to the theoretical probability of the event.
- ▶ As you increase the number of rolls of a die, the proportion of times you see the face with five spots gets closer to  $1/6$ .





# Large Random Samples

If the sample size is large,  
then the empirical distribution of a random  
sample  
resembles the distribution of the population,  
with high probability.

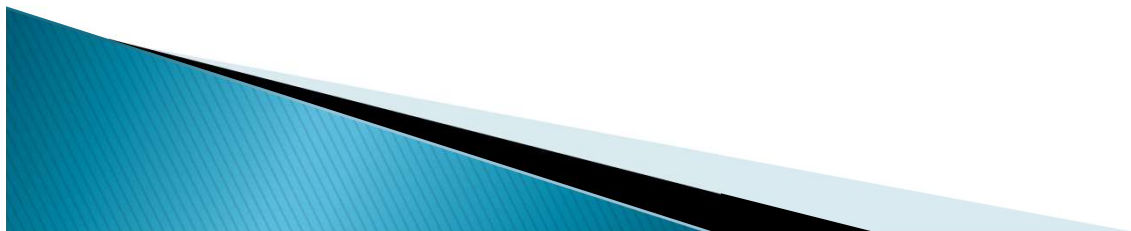
# Population exercise

- ▶ Normally, simple models like the difference equations or the differential equations are used to model population growth.
- ▶ However, these models track the number of individuals through time with a very simple growth factor from one generation to the next.
- ▶ The model in the exercise tracks each individual in the population and applies rules involving random actions to each individual.
- ▶ Such a detailed and much more computer-time consuming model can be used to see the effect of different policies. Using the results of this detailed model, we can (sometimes) estimate growth factors for simpler models so that these mimic the overall effect on the population size.



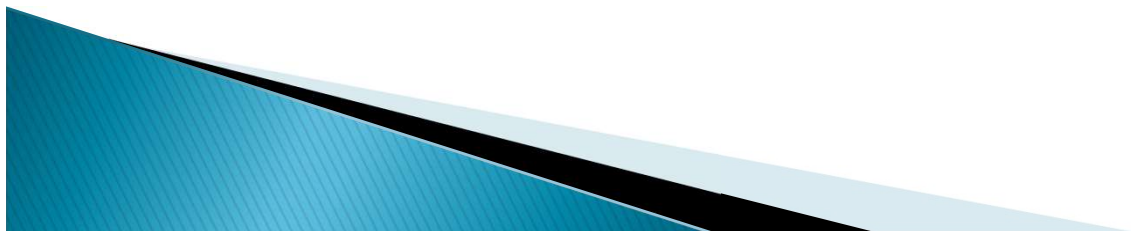
# File Objects

- ▶ `f = open(filename[, mode[, buffer_size])`
  - `mode` can be "r", "w", "a" (like C `stdio`); default "r"
  - append "b" for text translation mode
  - append "+" for read/write open
  - `buffer_size`: 0=unbuffered; 1=line-buffered; buffered
- ▶ methods:
  - `read([nbytes])`, `readline()`, `readlines()`
  - `write(string)`, `writelines(list)`
  - `seek(pos[, how])`, `tell()`
  - `flush()`, `close()`
  - `fileno()`



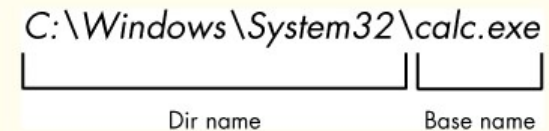
# \r, \n and \r\n

- ▶ \r (Carriage Return) – moves the cursor to the beginning of the line without advancing to the next line
- ▶ \n (Line Feed) – moves the cursor down to the next line without returning to the beginning of the line – *In a \*nix environment \n moves to the beginning of the line.*
- ▶ \r\n (End Of Line) – a combination of \r and \n



# os.path – Common pathname manipulations

- ▶ `os.getcwd()`
- ▶ `os.chdir(r'C:\Dersler\KOC\CMSF501\')`
- ▶ `os.mkdir('DENEME')`
- ▶ `os.path.abspath('.')`
- ▶ `os.path.join()` #to build paths in a way that will work on any operating system.
- ▶ `os.path.isfile(path)`
- ▶ `os.path.isdir(path)`
- ▶ `os.sep`



The diagram shows the path `C:\Windows\System32\calc.exe` with a bracket underneath it. The bracket is divided into two sections: the left section is labeled 'Dir name' and the right section is labeled 'Base name'.

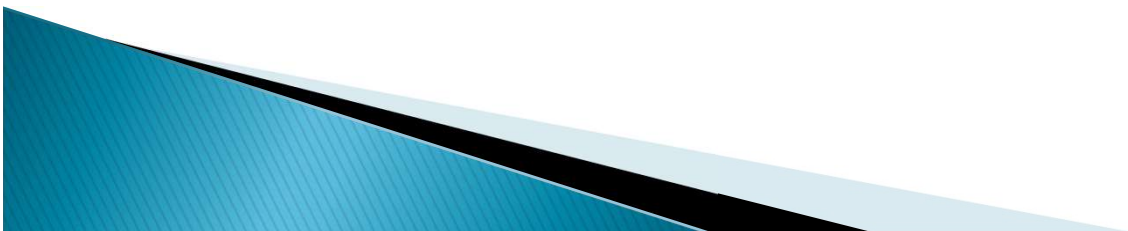
The full documentation for the `os.path` module is on the Python website at  
<http://docs.python.org/3/library/os.path.html>.



# os.path – Common pathname manipulations

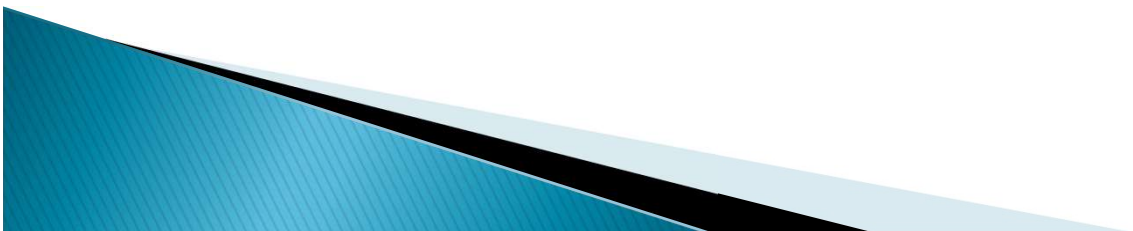
## os.path.splitdrive(*path*)

- ▶ Split the pathname *path* into a pair (drive, tail) where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, drive + tail will be the same as *path*.



# exec()

- ▶ `exec(open("file_name").read())`

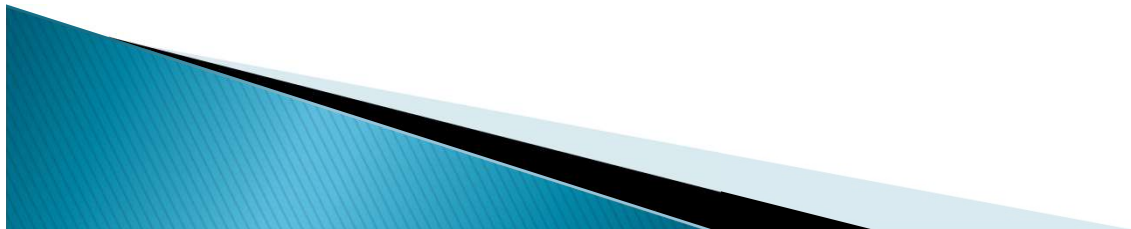


# Catching Exceptions

```
def foo(x):  
    return 1 / x
```

```
def bar(x):  
    try:  
        print foo(x)  
    except ZeroDivisionError, message:  
        print "Can't divide by zero:", message
```

```
bar(0)
```





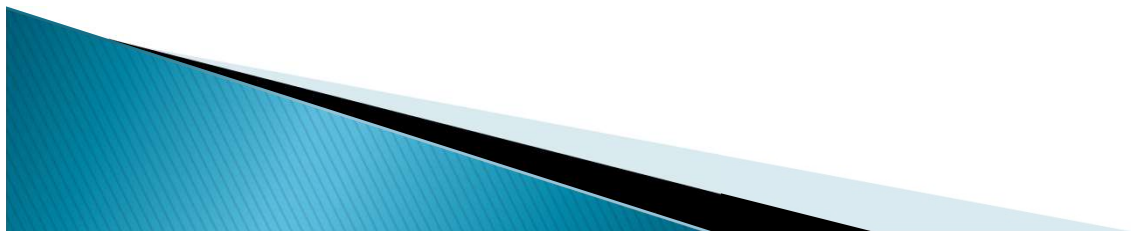
# Try-finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()      # always executed
print "OK" # executed on success only
```



# Raising Exceptions

- ▶ `raise IndexError`
- ▶ `raise IndexError("k out of range")`
- ▶ `raise IndexError, "k out of range"`
- ▶ `try:`  
    *something*  
`except: # catch everything`  
    `print "Oops"`  
    `raise # reraise`



# More on Exceptions

- ▶ User-defined exceptions
  - subclass Exception or any other standard exception
- ▶ Old Python: exceptions can be strings
  - WATCH OUT: compared by object identity, not ==
- ▶ Last caught exception info:
  - `sys.exc_info() == (exc_type, exc_value, exc_traceback)`
- ▶ Last uncaught exception (traceback printed):
  - `sys.last_type, sys.last_value, sys.last_traceback`
- ▶ Printing exceptions: traceback module

