DATA 601

Additional Material:
If-elif Statements
Loops: While and For
Builtin Iterators
CSV File Loading
Functions and Arguments

The if-else-if ladder with a nested compound

```
if condition-1:
            # do this
            # do this
elif condition-2:
    if condition-A:
            # do this
    else:
            # do this
elif condition-3:
            # do this
            # do this
else:
            # do this
            # do this
This line always done
```

Logical Operators in Python

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal
and	both must be true
or	one or both must be true
not	reverses the truth value

```
a   = True
if (a):             # True
if not (a):         # False
```

```
if (a == 2) or (b == 3):            # If a == 2 OR b == 3
if (a == 2) and (b == 3):           # If a == 2 AND b == 3
```

```
if x < y < z:  # Is (x < y) and (y < z)
id x < y > z   # Is fine.
```

```
x if condition else y
# which means: x if condition; y if not condition.
```

While and For Loops in Python

```
While Condition-1:
        if Condition-2:
                #do this
                break
        elif Condition-3:
                #do this
                continue
else:
        # do this
```

ends looping entirely

ends current loop and starts next

only works only if the while loop ends successfully

```
for loop_control_target_variable in sequence:
        # do this
        for loop_control_target_variable2 in sequence2:
                # do this
```

Zip

```
a = ['Jim','Liz','Eva','Adam','Sam']
b = [10,20,30,15,22]
z = zip(a,b)
d = {}
for t in z:
        d[t[0]] = t[1]
print(d)
% OUTPUT: {'Jim': 10, 'Liz': 20, 'Eva': 30, 'Adam': 15, 'Sam': 22}
```

zip() maps the similar index of multiple containers so that they can be used just using as single entity

Builtins: Iterators

next() returns the next item from the iterator.

We use the next() function to manually iterate through all the items of an iterator.

this value is returned if the iterator is exhausted (no items left)

```
next(iterator, default)
```

If you have any doubts about the iterator returning a value, this will return a default value at the end of the iterator. Obviously make sure it doesn't create an infinite loop.
If the default isn't given, it produces a warning at the end.

```
a = list(range(3))

it = iter(a)
for i in range(5):
    print(next(it, "missing"))
```

```
0
1
2
missing
missing
```

Standard input/output

You can redirect these, for example, at the command prompt:

Stdin from file:

```
python a.py < stdin.txt
```

Stdout to overwritten file:

```
python a.py > stdout.txt
```

Stdout to appended file:

```
python a.py >> stdout.txt
```

Both:

```
python a.py < stdin.txt > stdout.txt
```

Open

```
f = open("anotherfile.txt", xxxx)
```

Where `xxxx` is (from the docs):

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newlines mode (deprecated)

The default mode is 'r' (open for reading text, synonym of 'rt'). For binary read-write access, the mode 'w+b' opens and truncates the file to 0 bytes. 'r+b' opens the file without truncation.

Reading data

```
f = open("some_input_file.txt")
data = []
for line in f:
    parsed_line = str.split(line,",")
    data_line = []
    for word in parsed_line:
        data_line.append(float(word))
    data.append(data_line)
print(data)
f.close()
```

If you don't want to deal with closing, then use with, e.g

```
with open(" some_input_file.txt") as f:
```

CSV Reader Example

```
import csv
with open('./csv_files/addresses.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    firstnames, lastnames, streets, citys, states,zipcodes = [], [], [],[], [], []

    for row in readCSV:
        firstname, lastname, street = row[0], row[1], row[2]
        city, state, zipcode  = row[3], row[4], row[5]

        firstnames.append(firstname)
        lastnames.append(lastname)
        zipcodes.append(zipcode)

    print(firstnames)
    print(lastnames)
    print(zipcodes)
```

addresses.csv

John	Doe	120 Jefferson	Riverside	NJ	8075
Jack	McGinnis	220 hobo Av.	Phila	PA	9119
John "Da Man"	Repici	120 Jefferson	Riverside	NJ	8075
Stephen	Tyler	7452 Terrace	SomeTown	SD	91234
	Blankman		SomeTown	SD	298
Joan "the bone",	Anne Jet	9th, at Terra	Desert City	CO	123

Functions and *Args

```
def add(num1, num2):
    return num1 + num2
```

```
def add(num1 =0, num2=0):
    return num1 + num2
```

```
def funct1(num1, num2):
    return 2*num1 + num2
```

```
def sum (num1, num2, num3, num4):
        return num1 + num2 + num3 + num4
a = [1,2,3,4]
answer = sum(*a)
```

```
def sum (num1, num2, *others):
        sum = num1
        sum += num2
        for num in others:
                sum += num
        return sum
```

```
def sum(*nums):
        sum = 0
        for num in nums:
                sum += num
        return sum
```

Flexible parameterisation: **KWARGS

The same can be done with `**dict_name` (`**` is the dictionary unpacking operator), which will make a dictionary from unallocated kwargs:

```
def print_details (a, **details):
        first = details["first"]
        surname = details["surname"]
        print (first + " " + surname + " has " + a + " pounds")
```

```
print_details("5", first="George", surname="Formby")
```

Note that you can't use one of these to override other variables. If nothing is allocated, the dictionary is empty.

Global variables

Variables outside of functions in scripts are global: in theory they can be seen anywhere.
However, the rule about local assignments creating local variables undermines this.
To force a local assignment to a global variable, use the `global` keyword, thus:

```
b = 10
def a ():
        global b
        b = 20
        print(b)          # Prints 20.
print(b)                  # Prints 10.
a()
print(b)                  # Now prints 20 as the function
                          #   changes the global b.
```