Tuples, Dictionaries, Conditionals, Loops, And Exceptions

Tuples:

Tuples are like lists, but they are immutable. This means that the values a cannot change.

Tuples are useful as a convienent way to verify code hasn't been modified or destroyed. Libraries will often take advantage of the immutablility of the tuple

It comes down to how python allocates memeory to lists. Python assumes that if we set up a list, we might want to *append* values to the list. Tuples do not allow this

Tuples are more ergonomic ways or storing data.

`my_tuple = (1, 2.0, "hi")` will create a tuple

`print(my_tuple)[0]` will print 1

you can also use a tuple to break down assignment of variables

`Var1,Var2 = ("var1", "var2")` will return the indexed value of the tuple

Dictionary:

Unordered pairs of key-values

The keys are unique and immutable objects the values can change.

```
dict = {}
dict['name']='dave'
dict(name='dave',age='30')
dict = {'name':'dave','age','30')
dict.has_key('name') # this will check for a given key
name in a dictionary
dict.keys()
dict.items()
dict.values()
dict.get('age') # the python interpreter will not
crash if not found
del dict['name']
dict.clear()
dir(dict)

help(dict.has_key)
```

Exercise

Write a program that will take from the user a `service` and `port` as a `key-value` pair. If the user enters `0` the program will break out of the loop.

```
dic = {}
data = "1"
while data != "0":
    data=input("Please enter a service:")
    port=input("Please enter a port:")
    if data == "0":
        break
    dic[data]=port
print(dic)
f=open("ports.txt","w")
f.write(str(dic))
f.close()
```

Taking Input From the User

using the `input(whatToPrint)` only accepts numbers

using the `raw_input(whatToPrint)` only accepts strings

Conditionals - IF statements

Inside of programming languages there needs to be a way to tell the program which direction it needs to flow. Many times, there will be situations where based on criteria certain blocks of code need to be run.

the syntax for an `if` statement is:

```
if some_condition:
    what to do if True
else:
    what to do if False
```

If the condition is true Python will run whatever comes after the `if` statement, otherwise the `else` statement will be run.

Indentation and Python:

Indentation in python is big deal; it is the specified number of spaces (4 spaces or a TAB) are used in order to distinguish which of the following code is in the `if` statement.

elif

A shorter, more pythonic way to write `if` statements which will run after the previous test fails is the `elif` .

```
if x != 4:
    print "wow"
elif x > 10:
    print "bad"
```

there is **no limit** to the amount of `elif` statements you an run in a code block.

Operators

In order for the code to understand which conditions needs to be fulfilled, in order to know which blocks to run, *operators* allow programmers to build complex logical checks in their code.

Assignment Operators

Placing values inside of variables or incrementing/decrementing

Function	Operator	Example
assignment operator	`=`	`x = 2; print(x)`
increments a value/counter	`+=`	`x = 2;x+=; print(x)`
decrement a value/counter	`-=`	`x = 2;x-=; print(x)`

Arithmetic Operators

Classic math operators for calcuations and string manipulations

Function	Operator	Example
Addition	`+`	`x = 2; x + 2; print(x)`
Subtraction	`-`	`x = 2; x - 2; print(x)`
multiplication	`*`	`x = 2; x * 2; print(x)`
Power Of	`**`	`x = 2; x ** 2; print(x)`
Addition	`/`	`x = 6; x / 2; print(x)`
Modulus	`%`	`x = 7; x % 2; print(x)`

Comparative Operators

Checks to see how different two values or variables are.

Function	Operator	Example
Greater Than	`>`	`x = 2; y = 3; print(x>y)`
Less Than	`<`	`x = 2; y = 3; print(x<y)`

Function	Operator	Example
Greater Than or Equal to	`>=`	`x = 2; y = 3; print(x>=y)`
Less Than or Equal to	`<=`	`x = 2; y = 3; print(x>=y)`
Equal to	`==`	`x = 2; y = 3; print(x==y)`
Not Equal	`!=`	`x = 2; y = 3; print(x!=y)`

Using these basic operators we can build complex tests and checks to see specific situations handled elegantly in python

Logical Operators

Are used to operate on Boolean results and expressions.

Function	Operator	Example
logical **AND**	`and`	`x,y = 42; if x < 12 and y > 21: print("wo0o00ot")`
logical **OR**	`or`	`x,y = 42; if x < 12 or y > 21: print("wo0o00ot")`
logical **Xor**	`xor`	`x,y = 42; if x < 12 xor y > 21: print("wo0o00ot")`
logical **NOT**	`not`	`x,y = 42; if x < 12 not y > 21: print("wo0o00ot")`

These operators allow you to build Truth Tables with your data, testing for conditions to be met.

AND	True	False

AND	True	False
True	True	False
False	False	False
:----------	:----------:	-----:

OR	True	False
True	True	True
False	True	False
----------	:----------:	-----:

xor	True	False
True	False	True
False	True	False
:----------	:----------:	-----:

not	
True	False
False	True
:----------	:----------:

Exercises:

1. Write a program that gets two strings (`s1` and `s2`) from the user: a. Show the user the length of the strings b. Show which string is longer c. If they are equal, print they are equal
2.

Write a program that gets numbers from the user: a. Show the largest number. b. If the numbers are biggers than 10, print "larger than 10" c. Save the sum of the numbers in a new variable.

3. Write a program gets 5 numbers form a user and inserts them into a list: a. Show the largest number. b. Show a sorted list. c. Show the list reversed. d. Print the 5 numbers, and get a brand new set of numbers from the user.

4. Create a list and insert 5 numbers: a. Print to the user that only numbers can be input b. If the user will inser a number larger than 255, print and error

Loops:

Computers can not think about how repetitive a task is, this is one of their greatest advantages. Iteration is an extremely powerful tool, allowing a programmer to perform multiple tasks in series.

While Loops:

These loops run as long as the condition it is checking is `True`. While loops only run **IF** the condition is true.

```
while condition:
    do stuff
```

How to stop infinite loops Commands needs to be inserted into loop logic so that once a condition is met, it is performed and the program moves on. Unless you're designing an application that needs an inifinite loop, `break`, `continue`, and `pass` will be of great help.

Function	Operator
get out of the loop	`break
resume the loop if the condition isn't met	`continue`
do nothing, even if the condition is met	`pass`
----------	:----------:

For Loops

`for` loops allow us to map values from a list and perform actions on it.

```
for variable in list:
    do some_stuff
```

You can also instantiate a list in the for loop

```
for number in [1,2,3,]:
    print number
```

The **range** command will give you all the numbers in that range.

Exceptions:

There is a basic rule in Python; if the program tries to run a command that it can not handle, it will crash.

In order to keep the program from crashing in situations that can be handled programmatically, the concept of *error handling* and *exceptions* was invented.

an **exception** is is a feature that allows unexpexted errors during execution to be hanlded.

the `try` command and the `except` command allow code to run despite of errors.

```
try:
    some_function
except:
    some_other_function
```

The error message output from the handling can give you a lot of insight into what needs to be fixed for execution to continue.

Function	Description
exception	Base class for all exceptions
stopIteration	Raised when the next() method of an iterator does not point to any object
stopIteration	Raised when the next() method of an iterator does not point to any object
systemExit	raised by the `sys.exit()`
standardError	default for built in exceptions except stopIteration
ArithmeticError	all errors that occur for numeric calculation.
OverflowError	a calculation exceeds maximum limit for a numeric type.
FloatingPointError	when a floating-point calculation fails.

Function	Description
ZeroDivisionError	when division or modulo by zero
AssertionError	case of failure of the Assert statement.
AttributeError	case of failure of attribute reference or assignment.
EOFError	no input from either the raw_input() or input() and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	user interrupts program execution pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	an index is not found in a sequence.
KeyError	the specified key is not found in the dictionary
NameError	an identifier is not foundin the local or global namespace
UnboundLocalError	when trying to access a local variable in a function or method without having one assigned to it
EnvironmentError	all exceptions outsif of the python environment
----------	:----------:

```
try:
    here i write whatever i hope will run
except:
    this is where i print("i\'m going for coffee boss")
```

Function	Description
`else`	will only run if no exception happens
`finally`	will run at the end, regardless of condition
`raise`	print a custom error message
`exception as e`	insert an error message into variable `e`
---	:---:

```
try:
    a =0/0
except:
    print "Exception handled"
else:
    print "no exception handled"
finally:
    print "this would run anyway"
```

Exercises:

1. Write a script that uses error handling
2. Write a script that populates divide by zero error.
3. Write a script that will take a user input and print a number 4 times, handling the error if the user inputs a string.
4. Print all the numbers from 0 until the limit of your computer. When it crashes, rewrite your code to handle the error.
5. rewrite the last lesson's examples to handle errors.