**Python**

Before starting, in IDLE, go to Options and set the font size to 18.

* What is IDLE?
* Start it up. See ***IDLE start up slide***.
* Show how to install it yourself.
* 4+7
* Delimiters and proper nesting (with an aside on operator precedence)
* 4 + 7 \* 3
* (4 + 7) \* 3
* [3, 4, 5]
* [(4 + 7) \* 3, 4, 5]
* max([(4 + 7) \* 3, 4, 5])
* Give me 4 numbers:

\_\_ + \_\_ + \_\_ + \_\_

\_\_ # returns last computed value)

\_\_ / 4 # floating point divide

\_\_

* Do it again with the same 4 numbers (unless it came out even)

\_\_ + \_\_ + \_\_ + \_\_

\_\_ # returns last computed value

\_\_ // 4 # integer divide

\_\_

* Difference between integer and float
* Doing arithmetic:
* Basic operators: (p. 29) +, -, \*, /, //, % (modulus), \*\*
* Others available in special packages. More on that later.
* Modular arithmetic

4 + 7 % 3

5

>>> (4 + 7) % 3

2

* Control flow – simplest – sequence – grab my module sequence
* Control flow – more interesting - if statement:

Give me a number.

x = <your number> # comment on use of <> notation

 # x is a variable. Think of it as a box into which you can

 put something.

if x < 10:

 print("small") # comment your code

else:

 print("large")

* elif

Give me a number.

x = <your number>

if x < 10:

 print("small")

elif x > 100:

 print("big")

else:

 print("so-so")

* Boolean operators: We’ve used < and >. Perhaps more common one is equals, but we need to be careful:

>>> x = 7

>>> if x = 3:

SyntaxError: invalid syntax

Now do it with ==.

* Boolean connectors

>>> x = 2

>>> y = 5

>>> if x ==4 or y == 4:

 print ("Got it")

>>> x = 4

>>> if x ==4 or y == 4:

 print ("Got it")

 Got it

Also and, not

* Read input

x = input() # We could write this but how would the user know to start typing?

16

x # prints '16'. Notice the quotes we’ll come back to them.

x = input("Give me a number")

23

x

x = input("\nGive me a number ") # \n adds blank line at left, then we

 add blanks at right.

23

x

Cut and paste the elif code and do it again with the read-in value for x. It won’t work.

* Strings vs. numbers

x = int(x) # converts a string to an integer. Now it will work.

Cut and paste the elif code and do it again with x. It will work.

Note that the box (variable) that we are calling x can hold values of different types.

We now have three types: int, float, string.

* We can nest commands to do multiple things at once:

x = int(input("\nGive me a number ")) # note nested parens

* Note that variable names are case sensitive:

x = 4

x

X

* Annoying to have to type in from scratch every time we want to run our code. So we can define a function: (Change name of parameter to t to make point that it’s different from outside vars.)

def analyze(t):

 if t < 10:

 print("small")

 elif t > 100:

 print("big")

 else:

 print("so-so")

When we run this it creates the definition of analyze. It doesn’t run analyze.

We call x the input parameter. It’s a name for something that doesn’t YET have a value.

* Calling our function:

analyze(789)

* We can nest commands:

analyze(int(input("\nGive me a number "))) # note parens

* Saving what we’ve done:
	+ Make a new window. This creates a new thing called a module.
	+ Copy and paste analyze.
	+ Rename it so we can tell when we’re using it as opposed to the other one. Call it categorize.
	+ Now we need to save it. Where? Anything you store on this machine will go away when you log out. But you can save on your flash drive. In a pinch, you can also email your code to yourself. When you save your module, a file is created. You must give it a name. It needn’t be categorize, since in fact a module may contain many functions.
* How to run our module:
	+ Back to the main window.
	+ Type categorize(35) # The function will be undefined.
	+ Now run the module.
	+ Try categorize(35) again.
	+ When you run a def, it creates a dictionary entry for the name you are defining. Creating and saving a module is different from running it.
* A module that solves a problem may have a main program and one or more subprograms (functions). See slide ***main program***.
* Review what the interpreter is doing for us. It parses the strings we type, converts them into something that can run on the machine, then runs it.
* Can we work with more complex objects than simple numbers? Grab my average function.

def average(numbers):

 sum = 0

 for num in numbers:

 sum = sum + num

 return(sum/len(numbers))

Run the module. Note RESTART. Now let’s invoke it:

average([3, 4, 5])

* So what we’ve just done is to use a new control concept: a loop.
* Reminder about dealing with bad input. Try average with bad data. Then grab my averagewitherrorchecking:

def average(numbers):

 try:

 sum = 0

 for num in numbers:

 sum = sum + num

 return(sum/len(numbers))

 except:

 print("Only defined for list of numbers")

 return(0)

run it with [2, 3, ‘a’]

* Another kind of for loop. Now we want to walk through some lists that can be enumerated. grab my factorial module. Talk about starting at 0.

def factorial\_wrong(n):

 result = 1

 for j in range(n):

 result = result \* j

 return(result)

def factorial(n):

 result = 1

 for j in range(1,n+1):

 result \*= j

 return (result)

* Variables must be initialized. Load my file sum\_int. Run it on 8. Explain what went wrong. Fix it.
* Also explain here about the values that rangereturns. Type range(5). Type len(range(5)). Gets 5 items starting at 0.
* See ***Dawson website slide***: Getting files from the book’s site:

[www.courseptr.com/downloads](http://www.courseptr.com/downloads)

search on Dawson. Download py3e\_sorce.zip Must open in notepad, not double click.

* Nesting – In Python, physical indentation matters. Most other programming languages use explicit delimiters, but the effect is the same. See ***Java slide***.
* Objects and methods
	+ There are built-in ones. Later we could define our own.
	+ Modern programming languages provide large libraries so you don’t have to start from scratch.
* Strings

T.J. Watson 1943 quote:

quote = "I think there is a world market for maybe five computers."

>>> print(quote.upper())

I THINK THERE IS A WORLD MARKET FOR MAYBE FIVE COMPUTERS.

>>> print(quote.title())

I Think There Is A World Market For Maybe Five Computers.

>>> print(quote.swapcase())

i THINK THERE IS A WORLD MARKET FOR MAYBE FIVE COMPUTERS.

* Be careful about matching delimiters:

guess = "I think"

>>> guess

'I think'

>>> guess = 'I think'

>>> guess

'I think'

>>> guess = "I think'

* Strings are sequences of characters.

len("abc")

3

>>> s = "abcdef"

>>> len(s)

6

>>> s

'abcdef'

>>> s[0]

'a'

>>> s[5]

'f'

>>> s[6]

Traceback (most recent call last):

 File "<pyshell#46>", line 1, in <module>

 s[6]

IndexError: string index out of range

* You can write your own code to work with strings.
	+ Show the ***vowels output slide***.
	+ Show the vowels module:

def vowels(st):

 new = ""

 for i in range(len(st)):

 if st[i] == "a" or st[i] == "e" or st[i]== "i" or st[i] == "o" or st[i] == "u" or st[i] == "y":

 new = new + "\*"

 else:

 new = new + st[i]

 return(new)

* + Why do we need new? Why not just write st[i] = “\*”? Try it.
* Reminder: When you’re working with numbers, range may not work the way you think it should. Load my factorial file.
	+ Look first at factorial\_wrong. Try it on 4. Why is the answer 0? (Because range(5) produces the list (0, 1, 2, 3, 4).
	+ Now look at factorial. Try it on 4.
	+ Now try it on very long numbers. I’ve tried it up to 10000.
* **Hint for the Trust Fund Buddy assignment:** Mutable sequences. Also we’ll illustrate parallel lists. Read my file months.
	+ We first create the sequence of the correct length.
	+ Then fill in values. In a real program, we would probably read them from a file.
	+ Show month.
	+ Play with changing a value. Notice same point as factorial. Numbering starts at 0.
	+ Notice that we have two parallel lists. Let’s print the names of the months with n days. Invoke the function months.
* Another control construct: while loop: read my file fill\_buffer.

def fill\_buffer(ln):

 buffer = ""

 next = ""

 while len(buffer) < ln:

 next = input("Enter your next string: ")

 buffer = buffer + next

 return(buffer)

* + Try fill\_buffer(6) and give it “Austin” then “Texas” . It will be too long.
	+ So add, just before the return statement:

buffer = buffer[0:ln]

* I’d be lying to you if I made it seem that everything is always simple.
* Computers are not flexible in interpreting instructions in the same way people are. See ***“English” examples on slides***.
* Strange things can happen. For example, there are reserved words and function names that you cannot use as regular variable names. Take a look at fill\_buffer1:

def fill\_buffer1(len):

 buffer = ""

 next = ""

 while len(buffer + next) <= len:

 next = input("Enter your next string: ")

 buffer = buffer + next

 return(buffer)

* + Try it with 4. The only difference is that we tried to use len as a variable name.
	+ Now type print(keyword.kwlist) . These are words you can never use as identifiers.
* The logic of while loops:
	+ First look at ***while slide***. It will run forever. You might want this if you are the system prompt. But even then, perhaps there should be a way to quit.
	+ Then load my file whiles.py. The function while\_break uses break, but we don’t tell the user how to make it happen. Let’s fix it by changing the prompt.
	+ But when someone reads your code, they can’t see when the loop will end. Let’s fix that with while\_cond.
* If you do get into an infinite loop: Just kill the Python process.
* There are predefined modules that you may want to use. The import statement.

import math

>>> math.sqrt(4)

2.0

>>> math.sin(3.14/2) # Note that angles are in radians.

0.9999996829318346

* Sets.

s1 = {3, 5, 8}

>>> x = 3

>>> if x in s1:

 print("hit")

else:

 print("nope")

hit

* Dictionaries

mydict = {"beaker": "mug", "bonnet": "hood", "rubber": "eraser", "lorry": "truck"}

>>> mydict["bonnet"]

'hood'

I’ll need to say more about this when we get to Exclusive Networks because we’ll use a structure like:

>>> netdict = {"joe": ("cool guy", 5), "pete": ("geek", 10)}

>>> netdict["joe"][1]

5

>>>

* What’s really going on when we pass parameters to functions? Draw boxes to show what happens when the function is called. The value is copied into the box of the parameter.
	+ So, if it’s a basic value, it can’t change:

x = 5

def add1(num):

 num += 1

 return(num)

add1(x)

x # x won’t have changed

* + But if it is a value that is a pointer, while the pointer can’t change, the value it points to can:

x = [1, 3, 5]

def change\_first(s):

 s[0] += 1

 return(s)

change\_first(x)

x # x will have changed

* Files – this is always a tricky part. Read my vowels module.
	+ Assign a file name (with a path) to the variable file\_name
	+ Define the function that takes a file name and reads the text file and gets rid of vowels.

def file\_vowels(f1):

 my\_file = open(f1)

 st = my\_file.read()

 new = ""

 for i in range(len(st)):

 if st[i] == "a" or st[i] == "e" or st[i]== "i" or st[i] == "o" or st[i] == "u" or st[i] == "y":

 new = new + "\*"

 else:

 new = new + st[i]

 return(new)

 my\_file.close