INTRO TO PYTHON

ABOUT JULIA

- Software Engineer at Arizona State University
- Partner at A Place Called Up Consulting, LLC
- Born and raised in Germany
- Moved to Arizona in 2009 for grad school
- Software development enthusiast

MEET THE CLASS!

- What is your name?
- What programming experiences do you have?
- What do you hope to get out of this class?
- Fun fact about yourself.

WHAT WE WILL COVER TODAY

- Why Python?
- What is programming?
- Variables and arithmetic
- Statements and Error Messages
- Boolean Expressions and Conditionals
- Loops
- Functions
- Method calls
- Lists and dictionaries
- *Make a game!*

WHY PYTHON?

"Python in particular emerges as a near ideal candidate for a first programming language."

J. Zelle, http://mcsp.wartburg.edu/zelle/python/python-first.html

- Yet, used by professionals
- Readable, maintainable code
- Rapid rate of development
- Variety of applications

WHAT IS PYTHON USED FOR?

- System Administration (Fabric, Salt, Ansible)
- 3D animation and image editing (Maya, Blender, Gimp)
- Scientific computing (numpy, scipy)
- Web development (Django, Flask)
- Game Development (Civilization 4, EVE Online)

WHO IS USING PYTHON?

- Disney
- Dropbox
- Canonical and Red Hat
- Google
- YouTube
- NASA
- Eventbrite
- SurveyMonkey
- Reddit
- …

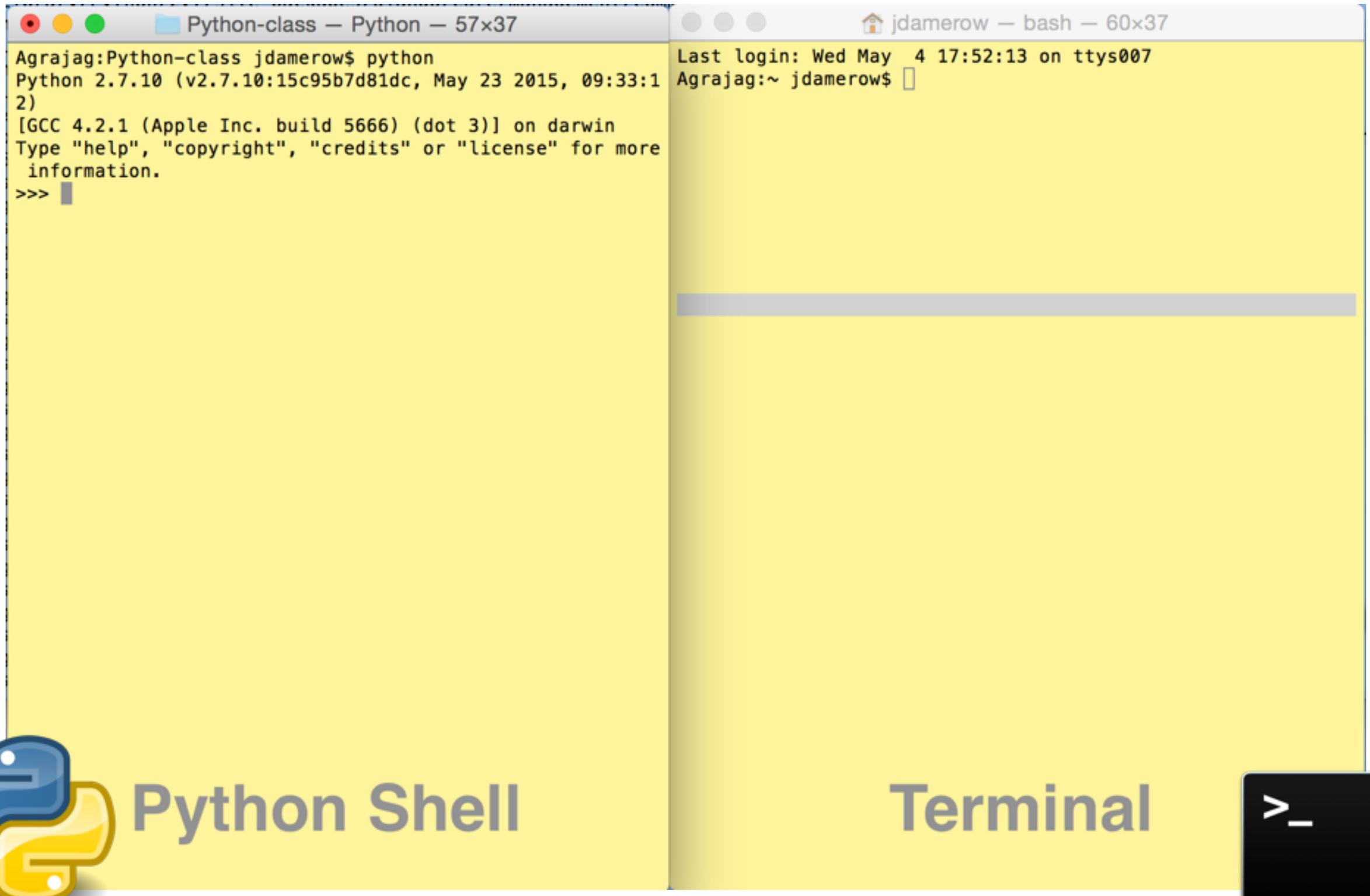
WHAT IS PROGRAMMING?

- Teaching the computer to do a task
- A program is made of one or more files of code, each of which solve part of the overall task.
- The code that you program is human readable but also needs to exist in a form that the computer can run directly. This form is not human readable.
- Don't focus on what's "under the hood" for now. We will "drive the car" first.

COMMAND LINE, PYTHON SHELL, TEXT EDITORS

Terminal	A program that has a command line interface and issues commands to the operating system.
Python Shell	A command line program that runs inside of the terminal, takes Python code as input, interprets it, and prints out any results.
Text Editor	A program that opens text files and allows the user to edit and save them. (Different than a word processor).

PYTHON SHELL VS. TERMINAL



EXAMPLE TEXT EDITORS

Linux	Gedit, Jedit, Kate
Mac OSX	TextMate, TextWrangler
Windows	Notepad++
All	Sublime Text, Vim, Emacs

LET'S DEVELOP IT

LET'S DEVELOP IT!

Working in the Python Shell

Open up your terminal and type: **`python`**

Follow along with the examples in the upcoming slides.
Just type them right in!

Feel free to explore, as well.
You will not accidentally break things!

VARIABLES AND ARITHMETIC

```
3 + 4
2 * 4
6 - 2
4 / 2
```

```
a = 2
b = 3
print(a + b)
c = a + b
print(c * 2)
```

```
a = 0
a = a + .5
print(a)
```

DATA TYPES

- Variables are used to store data
- Among other things, variables are used to represent something that can't be known until the program is run
- Data always has a "type"
- The type of a piece of data helps define what it can do
- The type can be found using: type()
- type() is a function. We call it by using parenthesis and pass it an object by placing the object inside the parenthesis

DATA TYPES (CONT.)

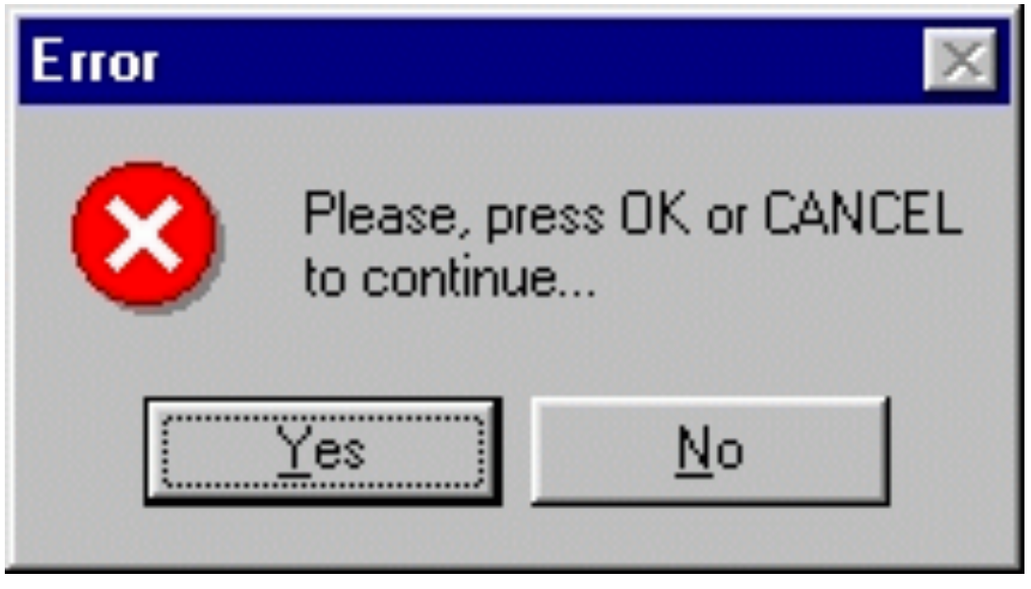
```
a = 4
print(type(a))
print(type(4))

print(type(3.14))

b = 'spam, again'
print(type(b))
print(type("But I don't like spam"))
```

DATA TYPES (CONT.)

- Data values can be used with a set of operators
- An "int" or "float" can be used with any of: +, -, *, /
- A "string" can be used with any of: +, *
- What happens if we try to use division or subtraction with a string?

```
print("Spam" - "am")
a = 'Spam and eggs'
print(a / 'hashbrowns')
print(a / 6)
```

ERRORS



ERRORS

- There are different kinds of errors that can occur.
  We've seen a few already.
- A "runtime error" results in an **Exception**, which has several types.
  Each type gives us some information about the nature of the error and how to correct it.
- One type of exception is a **SyntaxError**. This results when our code can not be evaluated because it is incorrect at a syntactic level.
  In other words, we are not following the "rules" of the language. E.g. in English, "run I" would be a syntax error (predicate before subject).
- Some other examples are the **TypeError** and **NameError** exceptions.

ERRORS (CONT.)

```
# SyntaxError - Doesn't conform to the rules of Python.
# This statement isn't meaningful to the computer
4spam)eggs(garbage) + 10

# NameError - Using a name that hasn't been defined yet
a = 5
print(b)
b = 10

# TypeError - Using an object in a way that its type does not support
'string1' - 'string2'
```

There are also semantic errors.

These are harder to catch because the computer can't catch them for us. E.g. "I was born in 1990."

LET'S DEVELOP IT

We'll practice what we've learned in the shell.

Review the slides on your computer and practice entering any commands you didn't fully understand before.

Ask the teacher, TAs, and students around you for help!

USING THE TERMINAL

USING THE TERMINAL

Try each of the following commands in turn:

Command	Short for	Description
`pwd`	Print working directory	Displays what folder you are in.
`ls`	List	Lists the files and folders in the current folder
`cd`	Change directory	Change to another folder. Takes the folder name as an argument. 'cd ..' goes up a directory
`cat`	Concatenate	Prints the contents of a file. Takes a filename as an argument

CREATING A FOLDER

We need a folder to save our work in.

The ->'s below indicate the expected output of the previous command.

```
pwd
-> /home/username
mkdir Projects
cd Projects
mkdir gdi-intro-python
cd gdi-intro-python
pwd
-> /home/username/Projects/gdi-intro-python
```

Now that the folders are made, we only have to use `cd Projects/gdi-intro-python` in the future.

THE TEXT EDITOR

Open your favorite text editor (e.g. Atom).

- Click "File", then "Open". Navigate to the gdi-intro-python folder we just created and click "Open". Then click "File", then "New File".
- In the text editor, enter the following:

```
print("I am a Python program!")
```

- Click "File", then "Save As...". Type "class1.py" and click "Save".
- Open a terminal and navigate to the `gdi-intro-python` folder.
  If you don't already have this folder open in a terminal.
- Type `python class1.py`
- You should see the terminal print "I am a Python program!"

USER INPUT

To obtain user input, use input().

Change the class1.py text to the following and run it again.

```
input_value = input("Enter a radius:")
radius = float(input_value)
area = 3.14159 * radius * radius
print("The area of a circle with radius " + input_value + " is:")
print(area)
```

The user's input is a string, so we use `float()` to make it a number.

LET'S DEVELOP IT

EXERCISE

Write a program that could be the beginning of a game. It should do the following:

```
>> Let’s play a game. What’s your name?
(wait for input)

>> Hi, (user’s name), you’re standing at the entrance of a labyrinth. Do you
want to go left, right, or straight?
(wait for input)

>> You chose to go (chosen direction) and find an ogre waiting. The ogre asks
you to guess its weight in pound:
(wait for input)

>> You tell the ogre you think he weighs somewhere between:
(input-10% to input+10%)

(wait for input before exiting)
```

BOOLEAN EXPRESSIONS AND CONDITIONALS

BOOLEAN EXPRESSIONS

We can tell the computer to compare values and return True or False. These are called Boolean expressions

- Test for equality by using ==.

  We can't use = because that is used for assignment.

- Test for greater than and less than using > and <.

```
a = 5
b = 5
print(a == b)
# Combine comparison and assignment
c = a == b
print(c)

print(3 < 5)
```

BOOLEAN EXPRESSIONS (CONT.)

The following chart shows the various Boolean operators

a == b	a is equal to b
a != b	a does not equal b
a < b	a is less than b
a > b	a is greater than b
a <= b	a is less than or equal to b
a >= b	a is greater than or equal to b

```
a = 3
b = 4
print(a != b)
print(a <= 3)
print(a >= 4)
```

Remember: Equals does not equal "equals equals"

CONDITIONALS

When we want different code to execute depending on certain criteria, we use a **conditional**.

We achieve this using **if statements**.

```
if x == 5:
    print('x is equal to 5')
```


We often want a different block to execute if the statement is false.

This can be accomplished using **else**.

```
if x == 5:
    print('x is equal to 5')
else:
    print('x is not equal to 5')
```

INDENTATION

In Python, a **block** begins when text is indented and ends when it returns to the previous indentation.

Let's look at the previous example again with a few minor changes and examine the meaning of its indentation

```
if x == 5:
    print('x is equal to 5')
    x_is_5 = True
    print('Still in the x == 5 block')
else:
    print('x is not equal to 5’)
    x_is_5 = False
    print('Still in the else block of x == 5')
print('Outside of the if or else blocks.')
print('x_is_5:')
print(x_is_5)
```

CHAINED CONDITIONALS

Conditionals can also be **chained**.

Chained conditionals use `elif` as an additional check after the preceding `if` predicate was False. For example

```
if x > 5:
    print('x is greater than 5')
elif x < 5:
    print('x is less than 5')
else:
    print('x is equal to 5')
```

CONDITIONALS VS. CHAINED CONDITIONALS

```
if x > 5:
   print("hallo")
if x >= 5:
   print("xxx")
else:
   print("bye")
```

```
if x > 5:
   print("hallo")
elif x >= 5:
   print("xxx")
else:
   print("bye")
```

NESTED CONDITIONALS

Conditionals can also be **nested**.

Nested conditionals occur inside of other conditionals and are indented over once more.

When the code block is complete, they are unindented

```
if x > 5:
    print('x is greater than 5')
    if x > 10:
        print('...it is also greater than 10')
    print('Done evaluating the x > 10 block')
print('Done evaluating the x > 5 block')
```

LET'S DEVELOP IT

Write a program that uses if statements to determine what to do given some user input.

The code below is an example:

```
health = 100
print("A vicious warg is chasing you.")
print("Options:")
print("1 - Hide in the cave.")
print("2 - Climb a tree.")
input_value = input("Enter choice:")
if input_value == '1':
    print('You hide in a cave.')
    print('The warg finds you and injures your leg with its claws')
    health = health - 10
elif input_value == '2':
    print('You climb a tree.')
    print('The warg eventually looses interest and wanders off')
```

ITERATION

ITERATION

- It is often useful to perform a task and to repeat the process until a certain point is reached.
- The repeated execution of a set of statements is called iteration.
- One way to achieve this, is with the while loop.

```
x = 10
while x > 0:
    print(x)
    x = x - 1
print('Done')
```

- The while statement takes a predicate, and as long as it evaluates to True, the code block beneath it is repeated.
- This creates a **loop**. Without the `x = x - 1` statement this would be an infinite loop.

WHILE LOOPS

Consider the following example that uses iteration to derive a factorial.

A factorial of a number is equal to that number * every positive integer less than that number. E.g. The factorial of 4 is 4 * 3 * 2 * 1, which equals 24.

```
input_value = input('Enter a positive integer:') # input('')
n = int(input_value)
result = 1
while n > 1:
    result = result * n
    n = n - 1
print("The factorial of " + input_value + " is:")
print(result)
```

This implementation does not work for negative numbers. Why?

FOR LOOPS

- It is also useful to loop through a collection of elements, visiting each one to do some work, then stopping once all elements are processed.
- This can be accomplished with a **for loop**.
- First, we need a collection. We create a **list** of numbers to loop over. This is called `numbers` in the following example:

```
numbers = [1, 3, 8]
for number in numbers:
    print("The current number is:")
    print(number)
```

FOR LOOPS (CONT.)

Let's examine the example carefully.

```
numbers = [1, 3, 8]
for number in numbers:
    print("The current number is:")
    print(number)
```

The for loop has three parts:

- The collection to loop over: `numbers`
- The name to give each element when the loop begins again: `number`
- The block of statements to execute with the element: The two `print` statements

LET'S DEVELOP IT

- Write a program that obtains user input like the last program
- However, this program should not exit until the user types "quit".
- Hint: A loop should help you

```
health = 100
print("A vicious warg is chasing you.")
print("Options:")
print("1 - Hide in the cave.")
print("2 - Climb a tree.")
input_value = input("Enter choice:")
if input_value == '1':
    print('You hide in a cave.')
    print('The warg finds you and injures your leg with its claws')
    health = health - 10
elif input_value == '2':
    print('You climb a tree.')
    print('The warg eventually looses interest and wanders off')
```

FUNCTIONS

FUNCTIONS

- A named section of code that performs a specific task.
- When one uses a function, one makes a **function call**.
- We have already made a function call when using the `print`, `type`, `int`, or `float` functions.

```
a = '3'
print(type(a))
a = float(a)
```

FUNCTION CALLS

```
a = 3
print(type(a))
```

- A function can take **arguments**.
- In the example above, the variable `a` is passed as an argument to the function `type`.
- Arguments can also be called **parameters**.

```
# Some more function call examples

int('32')
str(32)
```

FUNCTION DEFINITION

The following example is a **function definition**.
This allows us to create our own functions

```
def print_plus_5(x):
    print(x + 5)
```

The function definition has the following parts:

- The `def` keyword signifies we are defining a function.
- The name of the function being defined: `print_plus_5`.
- The arguments in parentheses: `x`.
- The function **body**, which is a block of indented code that executes when the function is called: `print x + 5`.

FUNCTION RETURNS

A function can also **return** a value.
To do this, one uses the **return** keyword.

```
def plus_5(x):
    return x + 5

y = plus_5(4)
```

- This allows us to call a function to obtain a value for later use.
  (Not the same as printing the value)
- In this example, the function call `plus_5(4)` evaluates to 9, and `y` is set to this value.
- To determine what a function will return, use the **substitution method**.
- If return is not used, the function returns **None**.

FUNCTIONS WITH NO ARGUMENTS

A function does not have to take arguments,
as in the following example:

```
def newline():
    print('')

newline()
# prints an empty line. Nothing is returned
```

This is useful when the function does some work but doesn't need any parameters. I.e. the function is intended to always do the same thing.

FUNCTIONS WITH MORE THAN ONE ARGUMENT

A function can also take more than one argument separated by commas. For example:

```
def find_rectangle_area(width, height):
    return width * height

area = find_rectangle_area(3, 4)
# area is set to the value 12
```

SCOPE

The **scope** of a variable is the area of code in which a variable is still valid and can be used.

Variables defined within a function can not be used elsewhere.

```
def get_triangle_area(base, height):
    rect_area = base * height
    return rect_area / 2.0

triangle_area = get_triangle_area(10, 20)

print(height)
# NameError
print(rect_area)
# NameError
```

IMPORT STATEMENTS

- The **import** statement allows us to use Python code that is defined in one file in a different file.
- Import statements can be used to import various specialized libraries - such as scikit or nympy - into our code.
- **Python Standard Library**: a published collection of useful functions for doing a specific type of coding. There are math, statistics, data science, and graphics libraries, to name a few types.
  https://docs.python.org/3.5/library/
- The **from** keyword allows us to only import parts of a Python file.

```
from math import sqrt
sqrt(2)
```

LET'S DEVELOP IT

LET'S DEVELOP IT

- Write a program that asks the user to guess a number between a given range, such as 1 to 10.
- The program should give the user hints such as "too high" or "too low". Alternatively, the hints might be "warm" or "cold" depending on how close they are to the number.
- The computer will need to have a random number for the user to guess:

```
#At the top of the file
from random import randint

# Use this line where you need to have a random number.
# (Hint, this is probably used before the user input loop)
random_number = randint(1, 10)
```

MORE WITH FUNCTIONS

MORE WITH FUNCTIONS

Functions can also call other functions.

You can use this to break up tasks
into small pieces that rely on others to do their work.

MORE WITH FUNCTIONS (CONT.)

```
from math import sqrt

def absolute_difference(value_a, value_b):
    return abs(value_a - value_b)

def get_hypotenuse(a, b):
    return sqrt(a ** 2 + b ** 2)

def get_area_rectangle(width, height):
    return width * height

def print_area_and_hypotenuse(x1, y1, x2, y2):
    width = absolute_difference(x1, x2)
    height = absolute_difference(y1, y2)
    area = get_area_rectangle(width, height)
    hypotenuse = get_hypotenuse(width, height)
    print('Area of the rectangle is:')
    print(area)
    print('The diagonal of the rectangle is:')
    print(hypotenuse)
```

FUNCTION COMPOSITION

Function composition is when the output
of one function acts as the input of another.

```
from math import sqrt

def find_distance(p1, p2):
    return abs(p1 - p2)

def get_hypotenuse(a, b):
    return sqrt(a ** 2 + b ** 2)

def print_hypotenuse(x1, y1, x2, y2):
    print('The diagonal of the rectangle is:')
    print(get_hypotenuse(find_distance(x1, x2), find_distance(y1, y2)))

print_hypotenuse(1,2,3,4)

# f(g(x))
# is the same as:
#     y = g(x)
#     f(y)
```

REMEMBER

- **Functions** are called by their name: `my_function()`

- They can be passed data in the form of parameters: `my_function(my_parameter)`

- They usually return some sort of value. This all happens explicitly.

METHOD CALLS

Methods are also called by name but are associated with an object.

Really, they are identical to functions except that the data passed into it is passed implicitly.

For example, the integers and strings we've been using have methods attached to them.

We can use the `dir()` function to see the methods of an object and `help()` to see what they do.

LET'S DEVELOP IT

- Open a Python shell and define a string variable.
- Use `dir(string_variable)` and the `help()` function to explore the various methods.
- **Hint**: Like functions, some methods take arguments and others don't.
- **Hint**: Use `help()` on a method.

  It will tell you the arguments to use and the expected behavior.
- **Hint**: Don't be afraid of errors.

  They seem to be in a foreign language but they are there to help you.
  Read them carefully.

LISTS

LISTS

- A list is an ordered collection of elements.
- In Python, a list is defined using [ ] with elements separated by commas, as in the following example.

```
words = ['list', 'of', 'strings']
```

- A list can, but doesn't have to be of all one type.

  A list of one type is **homogenous** as opposed to a list of multiple types, which is **heterogeneous**.

```
# heterogenous list
words = [0, 'list', 'of', 3, 'strings', 'and', 'numbers']
```

LIST METHODS

Lists have several methods, the most useful of which is append.

A list can be created as an empty list and have values added to it with append.

```
to_dos = []
to_dos.append('buy soy milk')
to_dos.append('install git')
print(to_dos)
```

Therefore, lists are **mutable**.

This means that a list can change values during the duration of a program.

ITERATION

Lists and many other collections are **iterable**.

Once defined, we can iterate on them,
performing an action with each element.

```
shipping_cost = 2.5
prices = [3, 4, 5.25]
costs = []

for price in prices:
    costs.append(price + shipping_cost)

for cost in costs:
    print(cost)
```

INDEXING

- An element can also be obtained from a list through **indexing**.
- This allows us to obtain an element without iterating through the entire collection if we just want one value.
- To index on a collection, follow it immediately with `[index]`. (index here is a number, variable or expression)

```
numbers = [10, 20, 30]
print(numbers[0])
```

INDEXING (CONT.)

Lists and other collections in Python are **zero indexed**.

This means that the number 0 refers to first element in the list.

```
to_dos = [
    'install git', 'read email', 'make lunch',
]
print(to_dos[0])
print(to_dos[1])

print(to_dos[len(to_dos) - 1])
```

An IndexError results if an index exceeds the length of the list minus 1.

DICTIONARIES

DICTIONARIES

A **dictionary** (sometimes called a "hashmap") is a collection of key/value pairs, defined with {}.

```
menu_categories = {
    'food': 'stuff you eat',
    'beverage': 'stuff you drink',
}
```

Think of words in a dictionary.

The words are keys and the definitions are values.

This dictionary would be indexed with strings such as 'food' and 'beverage' instead of integers like in a list.

INDEXING ON DICTIONARIES

- Dictionaries aren't literally just for definitions. They represent a group of mappings. A mapping might be: menu items -> costs.
- We can also index on dictionaries.
- The most common indexes are strings, but they can be whatever type the keys are.

```
menu = {
    'tofu': 4,
}

tofu_cost = menu['tofu']
```

Indexing on a key that doesn't exist results in a KeyError.

If you aren't certain a key is present, you can use the `get` method.

DICTIONARY METHODS

Some of the most essential methods are `keys`, `values` and `items`.

```
menu = {
    'tofu': 4,
    'pizza': 8,
    'baguette': 3,
}

print(menu.keys())
print(menu.values())
print(menu.items())
print(menu.get('pizza'))
print(menu.get('water'))
print(menu.get('juice', 5))
```

`get` will return **None** if the key isn't present or a default value if provided.

THE IN OPERATOR

- The `in` operator is used to determine if an element is in a given collection.
- For dictionaries, the keys are searched for the element.

```
color = [255, 255, 0]
if 0 in color:
    print('0 is in the color')

menu = {'tofu': 4}
print('tofu' in menu)

names = ['Mary', 'Martha', 'George']
george_present = 'George' in names
```

LET'S DEVELOP IT

Write a program that opens a text file and does some processing.

- The program should take a obstacle as input (e.g. ogre, bees, or wolf) and print what happens when you have a certain item (e.g. cake, sword, or balloons).
- It should return true, if the item protects, otherwise false.
- Your program should randomly decide if the item protects against the obstacle.
- The program should use at least one function to do its work and you should be able to import this function in a Python shell and call it with an obstacle and an item.

The next slide has some code and other resources that should help you get started

LET'S DEVELOP IT- EXAMPLE CODE

```
from random import randint

def does_item_protect(obstacle, item):
    # Your code goes here.
    # Your code should do something with the obstacle and item variables and
assign the value to a variable for returning
    # You could use two dictionaries to assign what happens when a certain
item does or does not protect agains an obstacle.


input_obstacle = input(“You encounter: ”)
input_item = input(“You have a: “)
protected = does_item_protect(input_obstacle, input_item)
# Display the answer in some meaningful way
```

LET'S DEVELOP IT FURTHER!

- Let's make a short game out of this!
- Let your player master 12 corners to the holy grail (loop over 10).
- At every corner randomly choose an obstacle and let the user choose an item.
- You can now call the method you just wrote.
- If you start with a health of 100, you could subtract 20 points every time the item does not protect.
- If the player has a health greater than 0 after 12 rounds, they win. If not, they loose.