MFIN 290 Programming and Data Analysis for Business

Dr. Fred Park
Paul Merage School of Business
UC Irvine

January 8th, 2020

Lecture 1

Course Goals

- Learn Programming (Python)
- Learn Data Analysis
- Above two in the context of business applications
- Be able to utilize these skills in business, finance, and beyond!

Motivation

Why programming and data?

We are currently in the 4th industrial revolution also known as the data revolution.

Motivation

Why programming and data?

Basic engineering skills and data analysis are being incorporated into nearly every field.

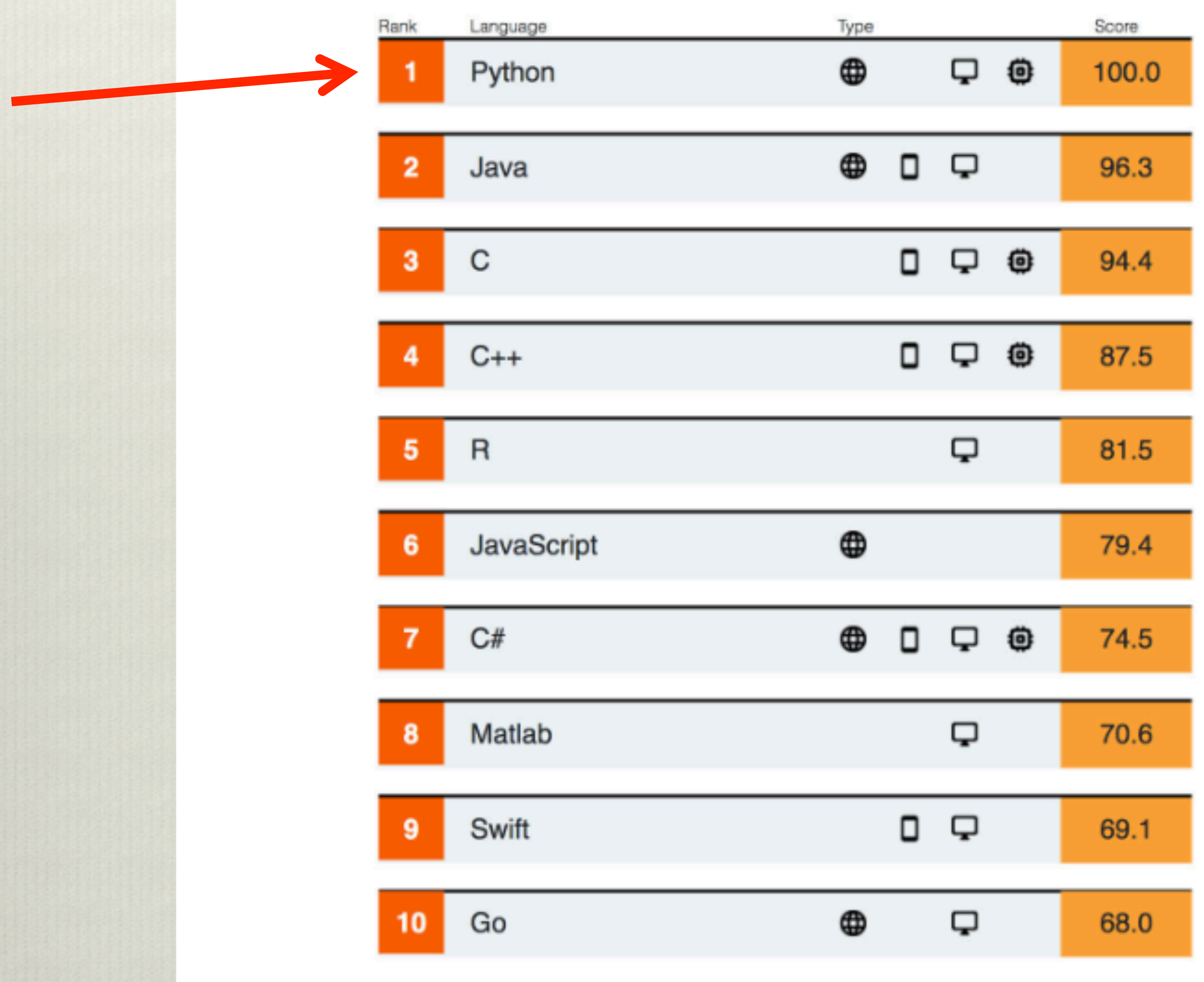
e.g. Fin Tech

Why Python?

- ❖ Python is becoming the standard language for data science/machine learning

- ❖ Most popular language in industry (IEEE 2019)

The Top Programming Languages of 2019



Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

Source: IEEE 2019 (Inst. of Electrical and Electronics Engineers)

Motivation

Python is:

- easy to learn
- syntax that follows normal language
- flexible
- fun!

Course Software

Hardest part of the course is installing the software!

- ❖ Anaconda package Python 3.7 version: most of the popular packages for python bundled. Includes Python https://www.anaconda.com/download/
- ❖ Pycharm IDE (Integrated Development Environment). Free with a student email.
- ❖ Other data manipulation packages/ML packages to be installed later in the course

Basic Elements of Python

- Python: Interpreted language (vs. compiled)
- A Python program often called a "script" is a sequence of definitions and commands
- The program is executed in a "shell." Shell is an interface: REPL: read, execute, print, loop
- Python Shell is basically an interactive interpreter. It's where your program runs
- Interpreter: reads code you write and converts to machine language which is of form 0's or 1's

e.g. 0100001110000111

Basic Elements of Python Cont’d

A **command** (also known as a **statement**) tells the interpreter to do something

For example: print(“Dodgers Rule!”)
Tells the interpreter to print: Dodgers Rule!

The commands:

```
print(“Dodgers Rule”,“and are #”, 1 )
print(“but not in New York!”)
print(“unless they win the World Series in New York!”)
```

print:

```
Dodgers Rule and are #1
but not in New York!
unless they win the World Series in New York!
```

Objects, Expressions, Numerical Types

- **Object:** core thing that python programs manipulate
- Objects have **types**: defines things programs can do with that type
- Types are either **scalar** or **non-scalar**:

1. scalar = indivisible (think of as atoms of python)
2. non-scalar = internal structure (a string for example)

Objects, Expressions, Numerical Types cont'd

4 types of scalar objects:

- ❖ int: represents integers. e.g. 1, -2, 2019
- ❖ float: represent real numbers e.g. 3.0, 3.14, -500.27
- ❖ bool: boolean values True or False
- ❖ None: type with single value (more later)

Objects, Expressions, Numerical Types cont’d

```
>>> print(4)
4
```

```
>>> type(4)
<class, ‘int’>
```

Type is ‘int’ for integer

```
>>> type(“Hello World!”)
<class, ‘str’>
```

Type is ‘str’ for string

```
>>> type(3.14)
<class, ‘float’>
```

Type is ‘float’ for floating point number

Objects, Expressions, Numerical Types cont’d

```
>>> type(3.14)
<class, ‘float’>
```

Type is ‘float’ for floating point number

What about?

```
>>> type(‘3.14’)
```

or

```
>>> type(“3.14”)
```

Objects, Expressions, Numerical Types cont'd

```
>>>type(3.14)
<class, 'float'>
```

Type is 'float' for floating point number

What about:

```
>>>type('3.14')
<class, 'str'>
```

Type is 'str' for string

Why?

Objects, Expressions, Numerical Types cont'd

>>>type(3.14)
<class, 'float'>
Type is 'float' for floating point number

What about:
>>>type('3.14')
<class, 'str'>
Type is 'str' for string

Why?
When you add "" or '' → string
e.g. 3.14 vs '3.14' vs "3.14"

Objects, Expressions, Numerical Types cont'd

- **Objects** and o**perators** combined → form **expressions**.
- These expressions evaluate to an object of some type.
- We call this the **value** of the expression.

e.g. 3+2 is the object 5 of type int

e.g. 3.0+2.0 is the object 5.0 of type float

Note: book uses value and object synonymously in the early chapters
e.g. 3.0+2.0 is the value 5.0 of type float

Once again: think of everything in Python as an object!

Objects, Expressions, Numerical Types cont’d

Some basic operators on objects of type int and float:

+ addition

- subtraction

/ division

% remainder upon division 5%3 gives 2

// integer division e.g. 3//2 gives 1

* multiplication

** power e.g. 2**3 gives 8

< less than

> greater than

<= less than or equal to

>= greater than or equal to

== is equal?

!= not equal

Some operators on ints and floats

- `i+j` is the sum of `i` and `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.
- `i-j` is `i` minus `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.
- `i*j` is the product of `i` and `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.
- `i//j` is integer division. For example, the value of `6//2` is the `int` `3` and the value of `6//4` is the `int` `1`. The value is 1 because integer division returns the quotient and ignores the remainder.
- `i/j` is `i` divided by `j`. In Python 2.7, when `i` and `j` are both of type `int`, the result is also an `int`, otherwise the result is a `float`. In this book, we will never use `/` to divide one `int` by another. We will use `//` to do that. (In Python 3, the `/` operator, thank goodness, always returns a `float`. For example, in Python 3 the value of `6/4` is `1.5`.)
- `i%j` is the remainder when the `int` `i` is divided by the `int` `j`. It is typically pronounced "i mod j," which is short for "i modulo j."
- `i**j` is `i` raised to the power `j`. If `i` and `j` are both of type `int`, the result is an `int`. If either of them is a `float`, the result is a `float`.
- The comparison operators are `==` (equal), `!=` (not equal), `>` (greater), `>=` (at least), `<`, (less) and `<=` (at most).

Variables

A **variable** is a name that refers to an object

```
>>> x = 7.0
>>>print(x)
7.0
```

```
>>> x     =     7
```

assignment goes from right to left

e.g. 7 is assigned to name x

Variables

Class Exercise timed 6 mins:

1. assign to the variable my_name your name and print out to the console

2. set numerical values to the two variables x and y and print out their sum and product

Variables

Python allows multiple assignments

```
>>> x, y = 2, 3
```

binds x to 2 and y to 3

what does the following do?:

```
>>> x, y = y, x
```

what does the following do?:

```
>>> a, b = 3.14, “Hello world!”
```

what does the following do?:

```
>>> from = 7
```

Variables

```
>>> from = 7
 File "<ipython-input-8-a60736d363b3>", line 1
    from = 7
         ^
SyntaxError: invalid syntax
```

Results in a syntax error!

Reason, there are certain reserved/key words in python:

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

These are words that have special meanings and cannot be used as variable names

Object Types Revisited

why is an int like 714 a scalar and a string like “Hello world!” non-scalar?

```
>>> x = 7
>>> y = “Hello world!”
>>> print(y[0])
H
>>> print(y[1])
e
>>> print(y[2])
l
>>> print(x[0])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'int' object is not subscriptable
```

non-scalars → underlying structure.

In this case, you can access individ. characters from the string!

Pycharm IDE

Pycharm is an example of an integrated development environment **(IDE)**

IDE:

- Text editor
- a shell with syntax highlighting
- integrated debugger

bug: error in the code
debugging: the act or process of removing bugs in code

Basic Input and Output

We already used the print() command
This is an example of output

What about input?

Python has an input() command

ex. prompt user to enter their name and then print it

```
>>>  nm = input(“please enter your name:\n”)
>? Dr. Park
>>> print(“Hi”, nm)
Hi Dr. Park!
```

Basic Input and Output

```
>>>  nm = input(“please enter your name:\n”)
>? Dr. Park
>>> print(“Hi”, nm)
Hi Dr. Park!
```

the \n represents newline

We will do more string formatting later in the quarter!

Question: how is the input stored? How can you check it?

Basic Input and Output

Class exercise timed 10 mins:

1. Write code to prompt you to enter 2 numbers sequentially
and then print out the sum of the two numbers

2. Write code to enter a word and then the number of times you want to see it printed out. Then print it out that many times.

Basic Input and Output

Class exercise timed 10 mins:

1. Write code to prompt you to enter 2 numbers sequentially and then print out the sum of the two numbers.

```
a, b = input(enter 1st #’), input(enter 2nd #’)
a, b = int(a), int(b)
print(“#1 + #2 =”,a+b)
```

2. Write code to enter a word and then the number of times you want to see it printed out. Then print it out that many times.

```
a = input('enter word’)
a = a+' ’
n = input('enter how many times you want to see it’)
n = int(n)
print(n*a)
```

Branching

Straight line programs: execute one statement after another until finished (kinda boring)

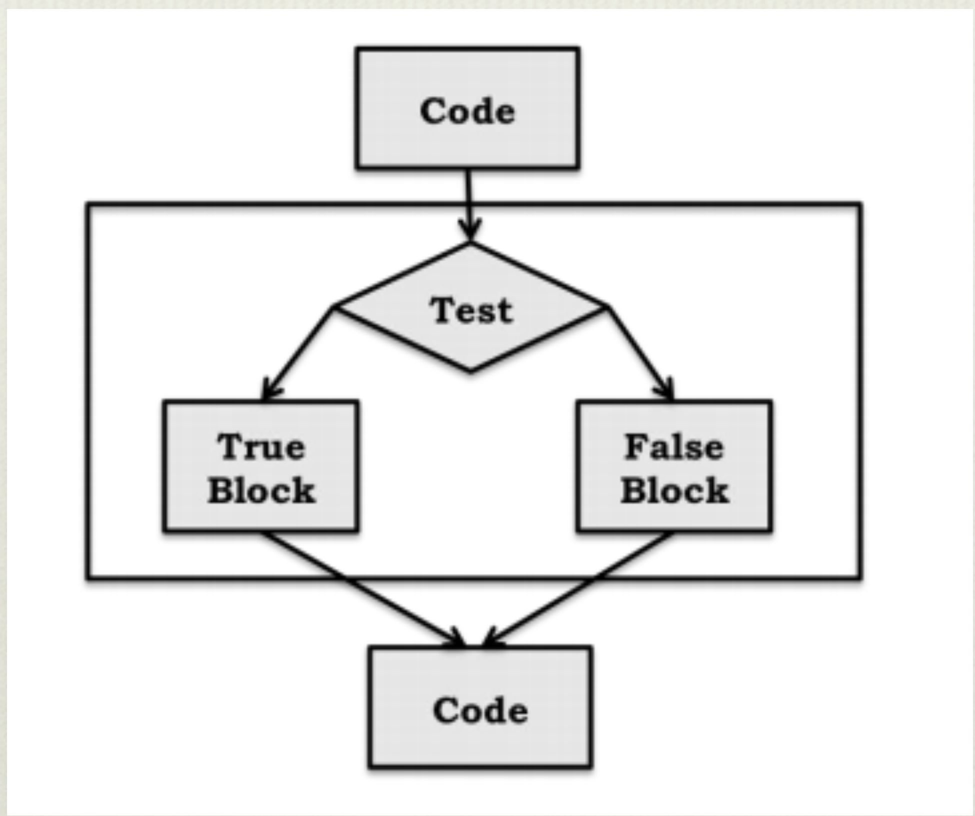
Branching programs → Branch depending on cases

Simplest branching program = conditional

Conditional statement has 3 parts:

1. A test → True or False
2. block of code if test is True
3. optional block of code if test is False

Branching



branching flow chart

Branching

In Python conditional has following form

```
if Boolean expression:
   block of code
else:
   block of code
```

Note: indentations in Python are semantically meaningful!

Branching

Example:

```
x=45
if x%2 == 0:
   print('even')
else:
   print('odd')
print('done with conditional')
```

outputs:

```
odd
done with conditional
```

Note:

x%2 == 0 → evaluates to True if remainder is 0

== → comparison

= → assignment e.g. x = 3

Branching

Example:

```
x=45
if x%2 == 0:
   print(‘even’)
else:
   print(‘odd’)
print(‘done with conditional’)
```

block corresponds to if statement

block corresponds to else statement

standalone block executed after conditional

Indentations are semantically meaningful!

Branching

Python general chained conditional has following form

```
if Boolean expression:
   block of code
elif Boolean expression:
   block of code
elif Boolean expression:
   block of code
.
.
.
else:
   block of code
```

Note: indentations in Python are semantically meaningful!
elif = else if
else is like a final catch basin if all above conditions are False

Nested Branching

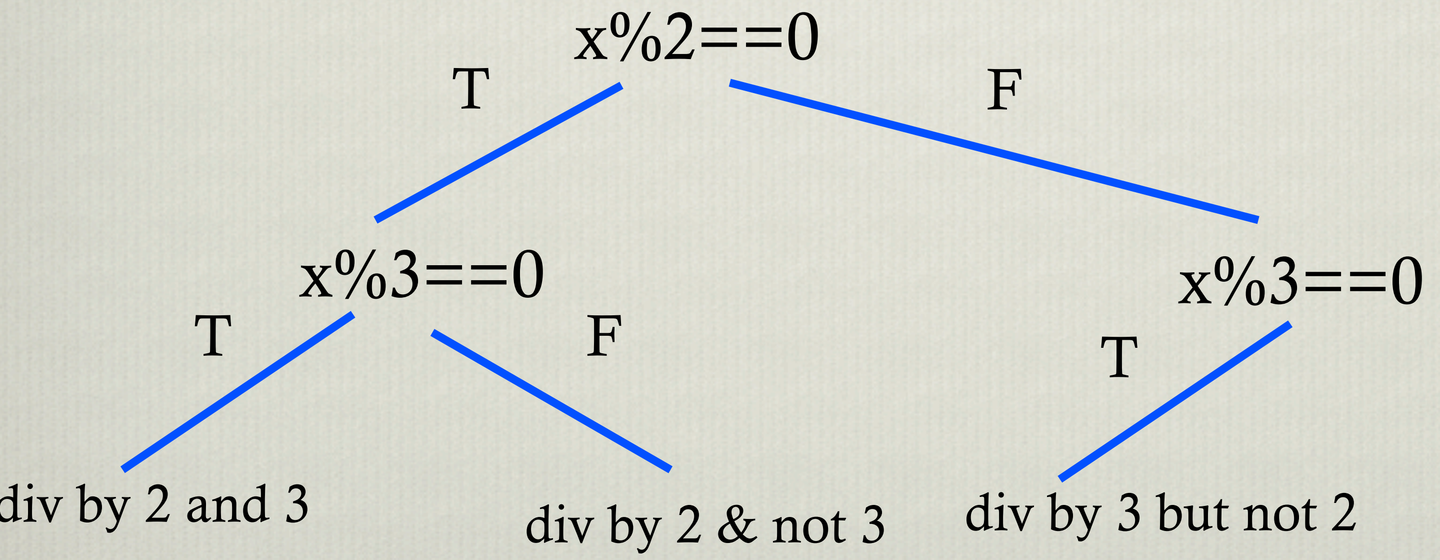
If the True or False block of a conditional contains another conditional → called **nested**

```
x=6
if x%2 == 0:
   if x%3 == 0:
      print('divisible by 2 and 3')
   else:
      print('divisible by 2 and not by 3')
elif x%3 == 0:
   print('divisible by 3 and not by 2')
```

note: elif means: else if

Nested Branching

```
x=45
if x%2 == 0:
   if x%3 == 0:
      print(‘divisible by 2 and 3’)
   else:
      print(‘divisible by 2 and not by 3’)
elif x%3 == 0:
   print(‘divisible by 3 and not by 2’)
```



Nested Branching Cont'd

Compound boolean expression

```
x,y,z = 1,2,3

if x < y and x < z:
   print('x is least')
elif y < z:
      print('y is least')
else:
      print('z is least')
```

$x < y$ and $x < z$ means x is less than both y and z

Nested Branching Cont'd

Class Exercise timed 10 mins:

Prompt a user to enter a score between 0 and 100. e.g. 95.
Then print out their grade based on the following rubric

>= 90 → grade = A
>= 80 → grade = B
>= 70 → grade = C
>= 60 → grade = D
< 60 → grade = F

Nested Branching Cont’d

Class Exercise timed 10 mins:

Prompt a user to enter a score between 0 and 100. e.g. 95.

Then print out their grade based on the following rubric:

- >= 90 → grade = A
- >= 80 → grade = B
- >= 70 → grade = C
- >= 60 → grade = D
- < 60 → grade = F

```
nm = input(’enter a # bet. 0 and 100:')
nm = int(nm)
if nm >= 90 and nm <= 100 :
    print('you got an A grade!')
elif nm >= 80 and nm <= 89:
    print('you got a B grade!')
elif nm >= 70 and nm <= 79:
    print('you got a C grade!')
elif nm >= 60 and nm <= 69:
    print('you got a D grade!')
else:
    print('you got an F grade!')
```

Strings

string objects have some special properties

ex.

```
>>> a = "Ricky Bobby"
>>> a[0]
R
>>> a[-1]
y
```

example of string indexing
In Python, indices start at 0.
a[-1] → gives last element
a[10]→ also gives last element

Question: What does a[5] give?

Strings

Slicing strings
Can extract parts of the string via slicing
a[start: stop: stride]
stride = spacing

ex. Type the following and what do you notice?

```
>>> a = '123456789'
>>> a[::]
?
>>> a[0::]
?
>>> a[:10:]
?
```

Strings

Slicing strings

Can extract parts of the string via slicing

a[start: stop: stride]

stride = spacing

ex. Type the following and what do you notice?

```
>>> a = '123456789'
>>> a[1::]
?
>>> a[1:10:]
?
>>> a[::2]
?
```

Strings

Slicing strings
Can extract parts of the string via slicing
a[start: stop: stride]
stride = spacing

exercise: from the string: a = '123456789'
In one line of code create a new string with only the odd numbers e.g. "13579"

exercise: do the same but evens

Strings

Slicing strings
Can extract parts of the string via slicing
a[start: stop: stride]
stride = spacing

exercise: from the string: a = '123456789'
In one line of code create a new string with only the odd numbers e.g. "13579"

```
>>> a[::2]
```

exercise: do the same but evens

```
>>> a[1::2]
```

Strings

Slicing strings
Can extract parts of the string via slicing
a[start: stop: stride]
stride = spacing

exercise: from the string: a = ‘123456789’
what does the following command do?
```
>>> a[1:-1:2]
```

Strings

Slicing strings

Can extract parts of the string via slicing

a[start: stop: stride]

stride = spacing

exercise: from the string: a = ‘123456789’

what does the following command do?

>>> a[1:-1:2]

‘2468’

Handling Errors in Input

Handling Exceptions

What if you prompt a user to enter an integer and they enter a string? You get an error.

```
>>> num = input('Please enter a #: '); num = int(num)
>? you got it!!
```

You get a ValueError!!

How do you bypass this?

Handling Errors in Input

How do you bypass this?

You can catch exceptions using the **try** and **except** clause

```
inp = input('Please enter an integer: ')
try:
    inp = float(inp)
    print("3 times your number =", 3*inp)
except:
    print('You did not enter an integer!')
```

If all goes well in the try block it skips except block
If error occurs in try block, goes to the except block

Fin 290, Class Exercise #1
Instructor: Dr. Fred Park

1. Write code that outputs your favorite short phrase.
2. Output 2 of your favorite words by joining them via a ','. Do the same by joining them with '+'. What's the difference?
3. Set your favorite word to the variable 'x'. Type in the shell prompt
   >> x*3
   What do you see? Do the same to the variable 'y' except set y = your favorite number.
4. Type in the shell prompt:
   >> pow(2, 3)
   what do you notice?
5. Whats the difference between the two phrases:
   - print("I'm happy today!")
   - print('I\'m happy today!')
6. What's the difference between the two divisions: '/' and '//'? What about '%'?
7. Are variables case sensitive? Why or why not?
8. How do you comment out part of your code? How's about for multilines of code?
9. Write code that swaps the values of two variables. Do this two ways.
10. Write code that outputs the largest of 3 distinct numbers.
11. Write code that outputs the largest odd integer from 2 distinct ones.