Intro to Python/Jupyter

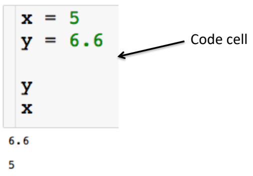
python™

Python Objects

- Variables are simply names that are used to keep track of information.
  - Variables are created when they are first assigned a value.
  - Variables must be assigned before they can be used.
- Variables will take the form of Python objects. We will use 3 different objects:
  - **Numbers**: integers, real number, etc …
  - **Strings**: ordered sequences of characters
  - **Lists**: ordered collection of objects
- All of the data we look at will take the form of numbers and strings.

Creating Variables

Let's create our first variables



```
x = 5
y = 6.6

y
x
```

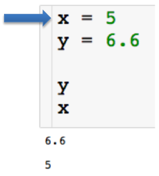
```
6.6
5
```

Code cell

The code executes from top to bottom

Creating Variables

Let's create our first variables



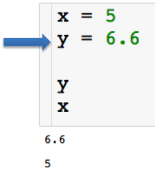
```
x = 5
y = 6.6

y
x
```

```
6.6
5
```

$x = 5$

Creating Variables

Let’s create our first variables. These variables with be numbers.



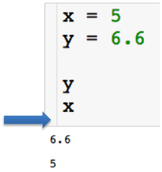
```
x = 5
y = 6.6

y
x
```

```
6.6

5
```

x = 5
y = 6.6

Creating Variables

Let’s create our first variables



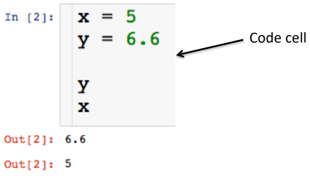
```
x = 5
y = 6.6

y
x
```

```
6.6

5
```

x = 5
y = 6.6

Creating Variables

Let’s create our first variables



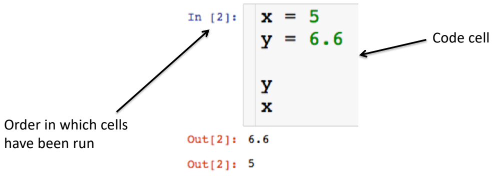
```
In [2]: x = 5
        y = 6.6

        y
        x
```

```
Out[2]: 6.6

Out[2]: 5
```

A couple of things about Jupyter

Creating Variables

Let’s create our first variables



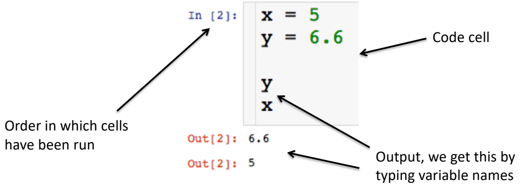
```
In [2]: x = 5
        y = 6.6

        y
        x
```

```
Out[2]: 6.6

Out[2]: 5
```

A couple of things about Jupyter

Creating Variables

Let's create our first variables



```
In [2]: x = 5
        y = 6.6

        y
        x

Out[2]: 6.6

Out[2]: 5
```

Code cell

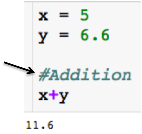
Order in which cells have been run

Output, we get this by typing variable names

A couple of things about Jupyter

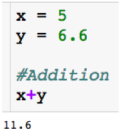
Arithmetic Operations

comment

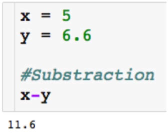


```
x = 5
y = 6.6

#Addition
x+y
11.6
```

Arithmetic Operations

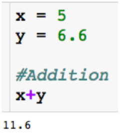


```
x = 5
y = 6.6

#Addition
x+y
11.6
```

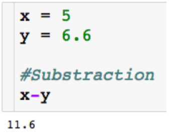


```
x = 5
y = 6.6

#Substraction
x-y
11.6
```

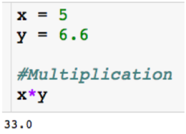
Arithmetic Operations



```
x = 5
y = 6.6

#Addition
x+y
11.6
```

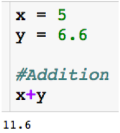


```
x = 5
y = 6.6

#Substraction
x-y
11.6
```



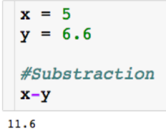
```
x = 5
y = 6.6

#Multiplication
x*y
33.0
```

Arithmetic Operations



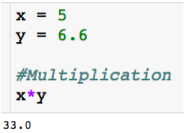
```
x = 5
y = 6.6

#Addition
x+y
```

```
11.6
```



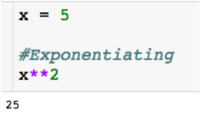
```
x = 5
y = 6.6

#Substraction
x-y
```

```
11.6
```



```
x = 5
y = 6.6

#Multiplication
x*y
```

```
33.0
```

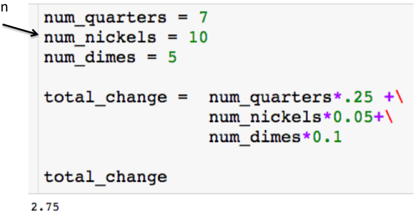


```
x = 5

#Exponentiating
x**2
```

```
25
```

Using Variables

Use description variable name



```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

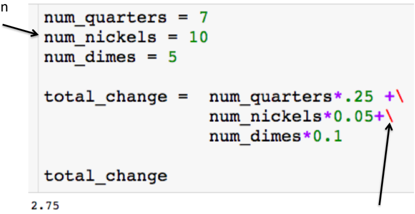
```
2.75
```

Rule for creating variable names:

- Be descriptive and separate words with underscore
- No spaces
- No punctuation other than underscore

Using Variables

Use description variable name



```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

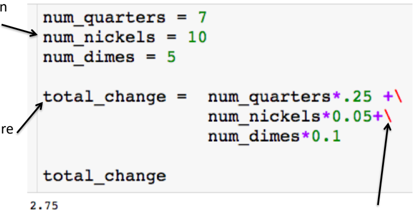
```
2.75
```

The backslash lets you continue your block of code on the next line.

Using Variables

Use description variable name

I can create variables that are a function of other variables

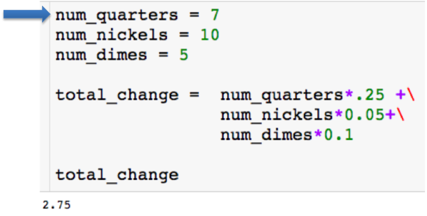


```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

```
2.75
```

The backslash lets you continue your block of code on the next line.

Using Variables

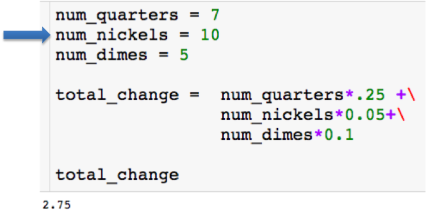


```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

```
2.75
```

num_quarters = 7

Using Variables

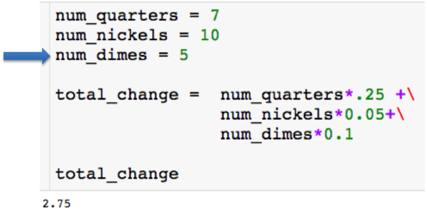


```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

```
2.75
```

num_quarters = 7
num_nickels = 10

Using Variables

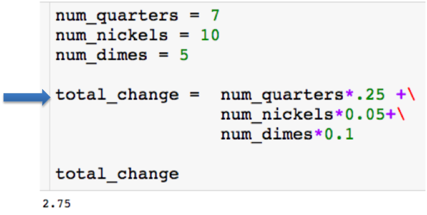


```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

```
2.75
```

num_quarters = 7
num_nickels = 10
num_nickels = 5

Using Variables

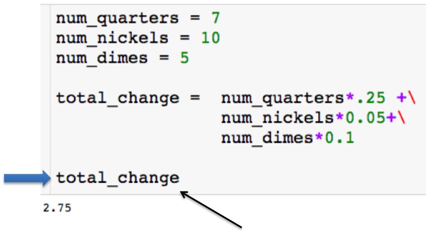


```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

```
2.75
```

num_quarters = 7
num_nickels = 10
num_nickels = 5
total_change = 2.75

Using Variables



```
num_quarters = 7
num_nickels = 10
num_dimes = 5

total_change =  num_quarters*.25 +\
                num_nickels*0.05+\
                num_dimes*0.1

total_change
```

```
2.75
```

num_quarters = 7
num_nickels = 10
num_nickels = 5
total_change = 2.75

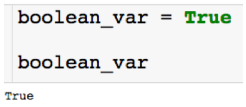
This just prints the value stored in the variable so we can see it.

Booleans

- The Boolean type can be viewed as numeric in nature because its values (True and False) are just customized versions of the integers 1 and 0.
- The True and False behave in the same way as 1 and 0, they just make the code more readable.
- Let us check if specified conditions are true

Booleans

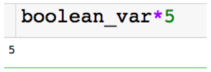
- Creating boolean variable:



```
boolean_var = True

boolean_var
```

```
True
```

- Note that the boolean does behave exactly like a 1:

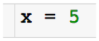


```
boolean_var*5
```

```
5
```

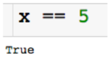
Conditional Tests

- Sets the variables x equal to 5.



```
x = 5
```

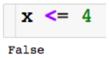
- Asks if x is equal to 5. Returns boolean.



```
x == 5
```

```
True
```

- Asks if x is less than or equal to 4. Returns boolean.



```
x <= 4
```

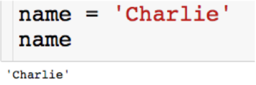
```
False
```

Strings

- Python strings are an ordered collection of characters (usually these characters will be letters and numbers) used to represent text.
- String are created by placing single or double quotation marks around a sequence of characters.
- Strings support the following operations
  - concatenation (combining strings)
  - slicing (extracting sections)
  - Indexing (fetching by offset)
  - the list goes on ….

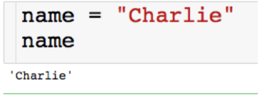
Strings

Let’s create our first strings



```
name = 'Charlie'
name
```

```
'Charlie'
```



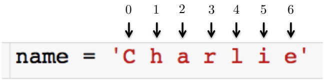
```
name = "Charlie"
name
```

```
'Charlie'
```

- You can create a string with either single or double quotes.
- There is a left to right ordering that we will explore on the next slide

Indexing Strings

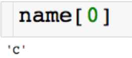
We can access the characters of the string through their **index**



```
name = 'C h a r l i e'
```

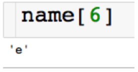
(pretend there aren’t spaces between the letters)

Slicing single characters through e index:



```
name[0]
```

```
'C'
```

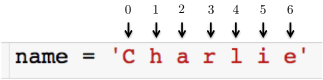


```
name[6]
```

```
'e'
```

Slicing Strings

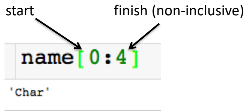
We can access the characters of the string through their **index**



```
name = 'C h a r l i e'
```

(pretend there aren’t spaces between the letters)

Slicing contiguous characters:



start

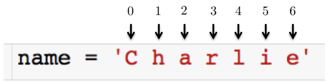
finish (non-inclusive)

```
name[0:4]
```

```
'Char'
```

Slicing Strings

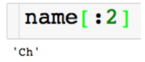
We can access the characters of the string through their **index**



```
 0 1 2 3 4 5 6
name = 'C h a r l i e'
```

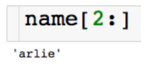
(pretend there aren't spaces between the letters)

Slicing contiguous characters:



```
name[:2]
'Ch'
```

If start index is left blank defaults to 0

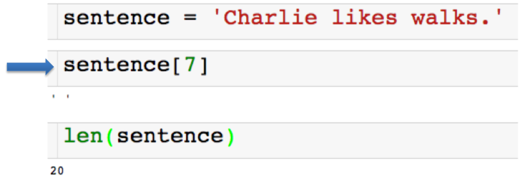


```
name[2:]
'arlie'
```

If end index is left blank defaults to end of the string

Slicing Strings

We can access the characters of the string through their **index**

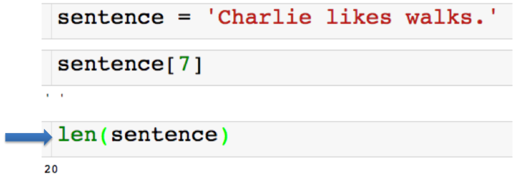


```
sentence = 'Charlie likes walks.'

sentence[7]
' '

len(sentence)
20
```

Spaces and punctuation count in the indexing of a string!

Slicing Strings

We can access the characters of the string through their **index**



```
sentence = 'Charlie likes walks.'

sentence[7]
' '

len(sentence)
20
```

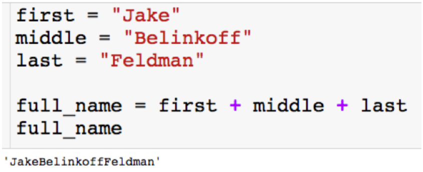
Returns the number of characters in the string

String Concatenation

- I can combine strings using the + operator.
- So the + operator between two numbers add them and the + operator between two strings concatenates them! This is called **polymorphism.**

String Concatenation

- I can combine strings using the + operator.
- So the + operator between two numbers add them and the + operator between two strings concatenates them! This is called **polymorphism.**



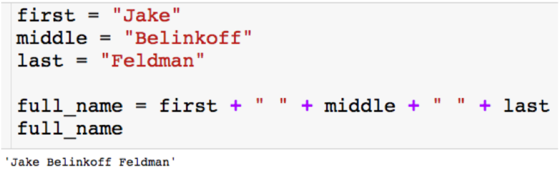
```
first = "Jake"
middle = "Belinkoff"
last = "Feldman"

full_name = first + middle + last
full_name
```

```
'JakeBelinkoffFeldman'
```

- If we want a space, we have to say so.

String Concatenation

- I can combine strings using the + operator.
- So the + operator between two numbers add them and the + operator between two strings concatenates them! This is called **polymorphism.**



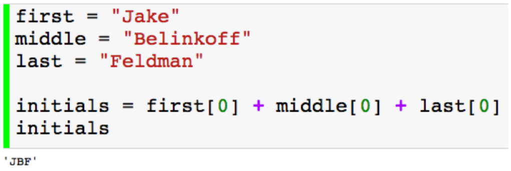
```
first = "Jake"
middle = "Belinkoff"
last = "Feldman"

full_name = first + " " + middle + " " + last
full_name
```

```
'Jake Belinkoff Feldman'
```

- With the space

String Concatenation

- I can combine strings using the + operator.
- So the + operator between two numbers add them and the + operator between two strings concatenates them! This is called **polymorphism.**



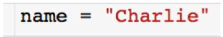
```
first = "Jake"
middle = "Belinkoff"
last = "Feldman"

initials = first[0] + middle[0] + last[0]
initials
```

```
'JBF'
```

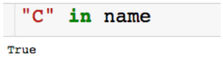
- Another example

Using In

- We can use the keyword in to check if a string is contained in another string.

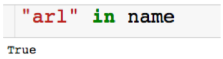


```
name = "Charlie"
```



```
"C" in name
```

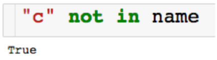
```
True
```



```
"arl" in name
```

```
True
```

- There is also a not in:



```
"c" not in name
```

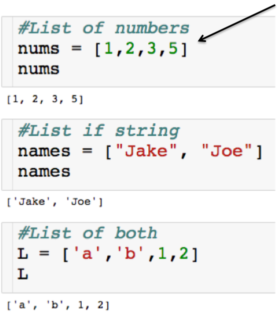
```
True
```

Lists

- Ordered collection of arbitrary objects.
  - There is a left to right ordering (just like string).
  - Can contain numbers, string, or even other lists.
- Elements accessed by offset.
  - You can fetch elements by index (just like string).
  - You can also do slicing and concatenation.
- Variable in length and arbitrarily nestable.
  - Lists can grow and shrink in-place.
  - You can have lists of lists of lists…

Lists

- Lets create our first lists



```
#List of numbers
nums = [1,2,3,5]
nums
```

```
[1, 2, 3, 5]
```

```
#List if string
names = ["Jake", "Joe"]
names
```

```
['Jake', 'Joe']
```

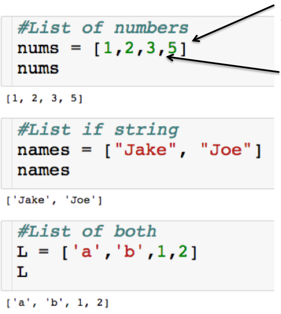
```
#List of both
L = ['a','b',1,2]
L
```

```
['a', 'b', 1, 2]
```

Elements enclosed in square brackets

Lists

- Lets create our first lists



```
#List of numbers
nums = [1,2,3,5]
nums
```

```
[1, 2, 3, 5]
```

```
#List if string
names = ["Jake", "Joe"]
names
```

```
['Jake', 'Joe']
```

```
#List of both
L = ['a','b',1,2]
L
```

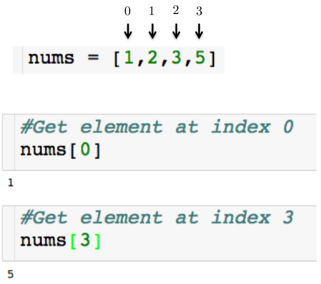
```
['a', 'b', 1, 2]
```

Elements enclosed in square brackets.

Elements separated by commas.

Indexing Lists

- Indexing for lists is very similar to strings



```
nums = [1,2,3,5]
```

```
#Get element at index 0
nums[0]
```

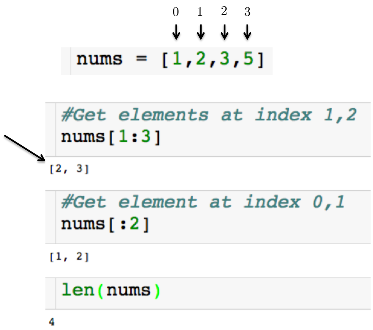
```
1
```

```
#Get element at index 3
nums[3]
```

```
5
```

Slicing Lists

- Slicing for lists I also very similar to strings



```
nums = [1,2,3,5]
```

Returns lists

```
#Get elements at index 1,2
nums[1:3]
```

```
[2, 3]
```

```
#Get element at index 0,1
nums[:2]
```

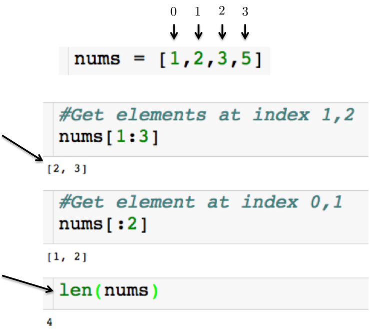
```
[1, 2]
```

```
len(nums)
```

```
4
```

Slicing Lists

- Slicing for lists I also very similar to strings



```
nums = [1,2,3,5]
```

Returns lists

```
#Get elements at index 1,2
nums[1:3]
```

```
[2, 3]
```

```
#Get element at index 0,1
nums[:2]
```

```
[1, 2]
```

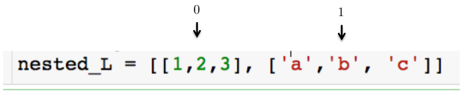
Returns # of
elements in list

```
len(nums)
```

```
4
```

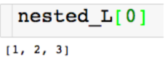
Nested Lists

- Creating a nested list:



```
nested_L = [[1,2,3], ['a','b', 'c']]
```

- There are two elements in the list nested_L.
  - There is a list of numbers in index 0.
  - There is a list of string of index 1.

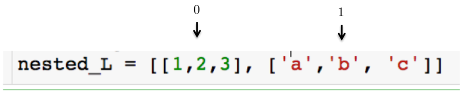


```
nested_L[0]
```

```
[1, 2, 3]
```

Indexing Nested Lists

- Creating a nested list:

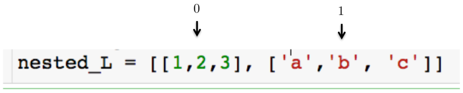


```
nested_L = [[1,2,3], ['a','b', 'c']]
```

- How do I pick out the 2 in the first list?

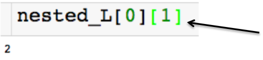
Indexing Nested Lists

- Creating a nested list:



```
nested_L = [[1,2,3], ['a','b', 'c']]
```

- How do I pick out the 2 in the first list?
  - First pick out the list of numbers, then from that pick out the



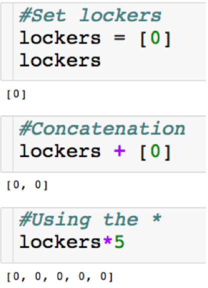
```
nested_L[0][1]
```

```
2
```

Stack the indexing

Polymorphism with Lists

- The + and * operator work on lists as well!



```
#Set lockers
lockers = [0]
lockers
```

```
[0]
```

```
#Concatenation
lockers + [0]
```

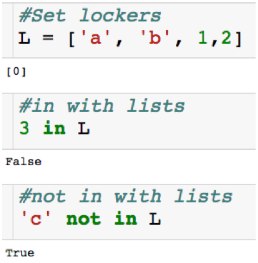
```
[0, 0]
```

```
#Using the *
lockers*5
```

```
[0, 0, 0, 0, 0]
```

Using in with Lists

- Keywords in and not in work with lists as well.



```
#Set lockers
L = ['a', 'b', 1,2]
```

```
[0]
```

```
#in with lists
3 in L
```

```
False
```

```
#not in with lists
'c' not in L
```

```
True
```