Numbers and more in Python!

In this lecture, we will learn about numbers in Python and how to use them.

We'll learn about the following topics:

```
1.) Types of Numbers in Python
2.) Basic Arithmetic
3.) Differences between classic division and f
loor division
4.) Object Assignment in Python
```

Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

Examples	Number "Type"
1,2,-5,1000	Integers
1.2,-0.5,2e2,3E2	Floating-point numbers

Now let's start with some basic arithmetic.

Basic Arithmetic

In [1]:

```
# Addition
2+1
```

Out[1]:

```
3
```

In [2]:

```
# Subtraction
2-1
```

Out[2]:

```
1
```

In [3]:

```
# Multiplication
2*2
```

Out[3]:

```
4
```

In [4]:

```
# Division
3/2
```

Out[4]:

```
1.5
```

In [5]:

```
# Floor Division
7//4
```

Out[5]:

```
1
```

Whoa! What just happened? Last time I checked, 7 divided by 4 equals 1.75 not 1!

The reason we get this result is because we are using "*floor*" division. The // operator (two forward slashes) truncates the decimal without rounding, and returns an integer result.

So what if we just want the remainder after division?

In [6]:

```
# Modulo
7%4
```

Out[6]:

```
3
```

4 goes into 7 once, with a remainder of 3. The % operator returns the remainder after division.

Arithmetic continued

In [7]:

```
# Powers
2**3
```

Out[7]:

```
8
```

In [8]:

```
# Can also do roots this way
4**0.5
```

Out[8]:

```
2.0
```

In [9]:

```
# Order of Operations followed in Python
2 + 10 * 10 + 3
```

Out[9]:

```
105
```

In [10]:

```
# Can use parentheses to specify orders
(2+10) * (10+3)
```

Out[10]:

```
156
```

Variable Assignments

Now that we've seen how to use numbers in Python as a calculator let's see how we can assign names and create variables.

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

In [11]:

```
# Let's create an object called "a" and assign it th
e number 5
a = 5
```

Now if I call *a* in my Python script, Python will treat it as the number 5.

In [12]:

```
# Adding the objects
a+a
```

Out[12]:

```
10
```

What happens on reassignment? Will Python let us write it over?

In [13]:

```
# Reassignment
a = 10
```

In [14]:

```
# Check
a
```

Out[14]:

```
10
```

Yes! Python allows you to write over assigned variable names. We can also use the variables themselves when doing the reassignment. Here is an example of what I mean:

In [15]:

```
# Check
a
```

Out[15]:

```
10
```

In [16]:

```
# Use A to redefine A
a = a + a
```

In [17]:

```
# Check
a
```

Out[17]:

```
20
```

The names you use when creating these labels need to follow a few rules:

```
1. Names can not start with a number.
2. There can be no spaces in the name, use _ i
nstead.
3. Can't use any of these symbols :'",<>/?|\
()!@#$%^&*~-+
4. It's considered best practice (PEP8) that n
ames are lowercase.
5. Avoid using the characters 'l' (lowercase l
etter el), 'O' (uppercase letter oh),
   or 'I' (uppercase letter eye) as single cha
racter variable names.
6. Avoid using words that have special meaning
in Python like "list" and "str"
```

Using variable names can be a very useful way to keep track of different variables in Python. For example:

In [18]:

```
# Use object names to keep better track of what's go
ing on in your code!
my_income = 100

tax_rate = 0.1

my_taxes = my_income*tax_rate
```

In [19]:

```
# Show my taxes!
my_taxes
```

Out[19]:

```
10.0
```

So what have we learned? We learned some of the basics of numbers in Python. We also learned how to do arithmetic and use Python as a basic calculator. We then wrapped it up with learning about Variable Assignment in Python.

Up next we'll learn about Strings!

Variable Assignment

Rules for variable names

- names can not start with a number
- names can not contain spaces, use _ intead
- names can not contain any of these symbols:

```
:'",<>/?|\!@#%^&*~-+
```

- it's considered best practice (PEP8 (https://www.python.org/dev/peps/pep-0008/#function-and-variable-names)) that names are lowercase with underscores
- avoid using Python built-in keywords like `list` and `str`
- avoid using the single characters `l` (lowercase letter el), `O` (uppercase letter oh) and `I` (uppercase letter eye) as they can be confused with `1` and `0`

Dynamic Typing

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

In [1]:

```
my_dogs = 2
```

In [2]:

```
my_dogs
```

Out[2]:

```
2
```

In [3]:

```
my_dogs = ['Sammy', 'Frankie']
```

In [4]:

```
my_dogs
```

Out[4]:

```
['Sammy', 'Frankie']
```

Pros and Cons of Dynamic Typing

Pros of Dynamic Typing

- very easy to work with
- faster development time

Cons of Dynamic Typing

- may result in unexpected bugs!
- you need to be aware of `type()`

Assigning Variables

Variable assignment follows `name = object`, where a single equals sign = is an *assignment operator*

In [5]:

```
a = 5
```

In [6]:

```
a
```

Out[6]:

```
5
```

Here we assigned the integer object `5` to the variable name `a`.
Let's assign `a` to something else:

In [7]:

```
a = 10
```

In [8]:

```
a
```

Out[8]:

```
10
```

You can now use a in place of the number `10`:

In [9]:

```
a + a
```

Out[9]:

```
20
```

Reassigning Variables

Python lets you reassign variables with a reference to the same object.

In [10]:

```
a = a + 10
```

In [11]:

```
a
```

Out[11]:

```
20
```

There's actually a shortcut for this. Python lets you add, subtract, multiply and divide numbers with reassignment using `+=`, `-=`, `*=`, and `/=`.

In [12]:

```
a += 10
```

In [13]:

```
a
```

Out[13]:

```
30
```

In [14]:

```
a *= 2
```

In [15]:

```
a
```

Out[15]:

```
60
```

Determining variable type with `type()`

You can check what type of object is assigned to a variable using

Python's built-in `type()` function. Common data types include:

- **int** (for integer)
- **float**
- **str** (for string)
- **list**
- **tuple**
- **dict** (for dictionary)
- **set**
- **bool** (for Boolean True/False)

In [16]:

```
type(a)
```

Out[16]:

```
int
```

In [17]:

```
a = (1,2)
```

In [18]:

```
type(a)
```

Out[18]:

```
tuple
```

Simple Exercise

This shows how variables make calculations more readable and easier to follow.

In [19]:

```
my_income = 100
tax_rate = 0.1
my_taxes = my_income * tax_rate
```

In [20]:

```
my_taxes
```

Out[20]:

```
10.0
```

Great! You should now understand the basics of variable assignment and reassignment in Python.
Up next, we'll learn about strings!

Strings

Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
3.) String Indexing and Slicing
4.) String Properties
5.) String Methods
6.) Print Formatting
```

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

In [1]:

```
# Single word
'hello'
```

Out[1]:

```
'hello'
```

In [2]:

```
# Entire phrase
'This is also a string'
```

Out[2]:

```
'This is also a string'
```

In [3]:

```
# We can also use double quote
"String built with double quotes"
```

Out[3]:

```
'String built with double quotes'
```

In [4]:

```
# Be careful with quotes!
' I'm using single quotes, but this will create an e
rror'
```

```
  File "<ipython-input-4-da9a34b3dc31>", line 2
    ' I'm using single quotes, but this will create
 an error'
       ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in `I'm` stopped the string. You can use combinations of double and single quotes to get the complete statement.

In [5]:

```
"Now I'm ready to use the single quotes inside a str
ing!"
```

Out[5]:

```
"Now I'm ready to use the single quotes inside a str
ing!"
```

Now let's learn about printing strings!

Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

In [6]:

```
# We can simply declare a string
'Hello World'
```

Out[6]:

```
'Hello World'
```

In [7]:

```
# Note that we can't output multiple strings this wa
y
'Hello World 1'
'Hello World 2'
```

Out[7]:

```
'Hello World 2'
```

We can use a print statement to print a string.

In [8]:

```
print('Hello World 1')
print('Hello World 2')
print('Use \n to print a new line')
print('\n')
print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

String Basics

We can also use a function called len() to check the length of a string!

In [9]:

```
len('Hello World')
```

Out[9]:

```
11
```

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets `[]` after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called `s` and then walk through a few examples of indexing.

In [10]:

```
# Assign s as a string
s = 'Hello World'
```

In [11]:

```
#Check
s
```

Out[11]:

```
'Hello World'
```

In [12]:

```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [13]:

```
# Show first element (in this case a letter)
s[0]
```

Out[13]:

```
'H'
```

In [14]:

```
s[1]
```

Out[14]:

Out[14]:

```
'e'
```

In [15]:

```
s[2]
```

Out[15]:

```
'l'
```

We can use a `:` to perform *slicing* which grabs everything up to a designated point. For example:

In [16]:

```
# Grab everything past the first term all the way to
the length of s which is len(s)
s[1:]
```

Out[16]:

```
'ello World'
```

In [17]:

```
# Note that there is no change to the original s
s
```

Out[17]:

```
'Hello World'
```

In [18]:

```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[18]:

```
'Hel'
```

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

In [19]:

```
#Everything
s[:]
```

Out[19]:

```
'Hello World'
```

We can also use negative indexing to go backwards.

In [20]:

```
# Last letter (one index behind 0 so it loops back a
round)
s[-1]
```

Out[20]:

```
'd'
```

In [21]:

```
# Grab everything but the last letter
s[:-1]
```

Out[21]:

```
'Hello Worl'
```

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

In [22]:

```
# Grab everything, but go in steps size of 1
s[::1]
```

Out[22]:

```
'Hello World'
```

In [23]:

```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[23]:

```
'HloWrd'
```

In [24]:

```
# We can use this to print a string backwards
s[::-1]
```

Out[24]:

```
'dlroW olleH'
```

String Properties

It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed or replaced. For example:

In [25]:

```
s
```

Out[25]:

```
'Hello World'
```

In [26]:

```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------
-----------------------
TypeError                                 Traceback
 (most recent call last)
<ipython-input-26-976942677f11> in <module>()
      1 # Let's try to change the first letter to
 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assign
ment
```

Notice how the error tells us directly what we can't do, change the item assignment!

Something we *can* do is concatenate strings!

In [27]:

```
s
```

Out[27]:

```
'Hello World'
```

In [28]:

```
# Concatenate strings!
s + ' concatenate me!'
```

Out[28]:

```
'Hello World concatenate me!'
```

In [29]:

```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

In [30]:

```
print(s)
```

```
Hello World concatenate me!
```

In [31]:

```
s
```

Out[31]:

```
'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

In [32]:

```
letter = 'z'
```

In [33]:

```
letter*10
```

Out[33]:

```
'zzzzzzzzzz'
```

Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

In [34]:

```
s
```

Out[34]:

```
'Hello World concatenate me!'
```

In [35]:

```
# Upper Case a string
s.upper()
```

Out[35]:

```
'HELLO WORLD CONCATENATE ME!'
```

In [36]:

```
# Lower case
s.lower()
```

Out[36]:

```
'hello world concatenate me!'
```

In [37]:

```
# Split a string by blank space (this is the defaul
t)
s.split()
```

Out[37]:

```
['Hello', 'World', 'concatenate', 'me!']
```

In [38]:

```
# Split by a specific element (doesn't include the e
lement that was split on)
s.split('W')
```

Out[38]:

```
['Hello ', 'orld concatenate me!']
```

There are many more methods than the ones covered here. Visit the Advanced String section to find out more!

Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

In [39]:

```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

Out[39]:

```
'Insert another string with curly brackets: The inserted string'
```

We will revisit this string formatting topic in later sections when we are building our projects!

Next up: Lists!

String Formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas'
points = 33

'Last night, '+player+' scored '+str(points)+' points.'  # concatenation

f'Last night, {player} scored {points} points.'          # string formatting
```

There are three ways to perform string formatting.

- The oldest method involves placeholders using the modulo `%` character.
- An improved technique uses the `.format()` string method.
- The newest method, introduced with Python 3.6, uses formatted string literals, called *f-strings*.

Since you will likely encounter all three versions in someone else's code, we describe each of them here.

Formatting with placeholders

You can use %s to inject strings into your print statements. The modulo % is referred to as a "string formatting operator".

In [1]:

```
print("I'm going to inject %s here." %'something')
```

```
I'm going to inject something here.
```

You can pass multiple items by placing them inside a tuple after the % operator.

In [2]:

```
print("I'm going to inject %s text here, and %s text
here." %('some','more'))
```

```
I'm going to inject some text here, and more text he
re.
```

You can also pass variable names:

In [3]:

```
x, y = 'some', 'more'
print("I'm going to inject %s text here, and %s text
here."%(x,y))
```

```
I'm going to inject some text here, and more text he
re.
```

Format conversion methods.

It should be noted that two methods `%s` and `%r` convert any python object to a string using two separate methods: `str()` and `repr()`. We will learn more about these functions later on in the course, but you should note that `%r` and `repr()` deliver the *string representation* of the object, including quotation marks and any escape characters.

In [4]:

```
print('He said his name was %s.' %'Fred')
print('He said his name was %r.' %'Fred')
```

```
He said his name was Fred.
He said his name was 'Fred'.
```

As another example, `\t` inserts a tab into a string.

In [5]:

```
print('I once caught a fish %s.' %'this \tbig')
print('I once caught a fish %r.' %'this \tbig')
```

```
I once caught a fish this       big.
```

```
I once caught a fish 'this \tbig'.
```

The `%s` operator converts whatever it sees into a string, including integers and floats. The `%d` operator converts numbers to integers first, without rounding. Note the difference below:

In [6]:

```
print('I wrote %s programs today.' %3.75)
print('I wrote %d programs today.' %3.75)
```

```
I wrote 3.75 programs today.
I wrote 3 programs today.
```

Padding and Precision of Floating Point Numbers

Floating point numbers use the format `%5.2f`. Here, `5` would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, `.2f` stands for how many numbers to show past the decimal point. Let's see some examples:

In [7]:

```
print('Floating point numbers: %5.2f' %(13.144))
```

```
Floating point numbers: 13.14
```

In [8]:

```
print('Floating point numbers: %1.0f' %(13.144))
```

```
Floating point numbers: 13
```

In [9]:

```
print('Floating point numbers: %1.5f' %(13.144))
```

```
Floating point numbers: 13.14400
```

In [10]:

```
print('Floating point numbers: %10.2f' %(13.144))
```

```
Floating point numbers:      13.14
```

In [11]:

```
print('Floating point numbers: %25.2f' %(13.144))
```

```
Floating point numbers:                     13.14
```

For more information on string formatting with placeholders visit https://docs.python.org/3/library/stdtypes.html#old-string-formatting (https://docs.python.org/3/library/stdtypes.html#old-string-formatting)

Multiple Formatting

Nothing prohibits using more than one conversion tool in the same print statement:

In [12]:

```
print('First: %s, Second: %5.2f, Third: %r' %('hi!',
3.1415,'bye!'))
```

```
First: hi!, Second:  3.14, Third: 'bye!'
```

Formatting with the `.format()` method

A better way to format objects into your strings for print statements is with the string `.format()` method. The syntax is:

```
'String here {} then also {}'.format('somethin
g1','something2')
```

For example:

In [13]:

```
print('This is a string with an {}'.format('insert'
))
```

```
This is a string with an insert
```

The .format() method has several advantages over the %s placeholder method:

1. Inserted objects can be called by index position:

In [14]:

```
print('The {2} {1} {0}'.format('fox','brown','quick'
))
```

```
The quick brown fox
```

2. Inserted objects can be assigned keywords:

In [15]:

```
print('First Object: {a}, Second Object: {b}, Third
 Object: {c}'.format(a=1,b='Two',c=12.3))
```

```
First Object: 1, Second Object: Two, Third Object: 1
2.3
```

3. Inserted objects can be reused, avoiding duplication:

In [16]:

```
print('A %s saved is a %s earned.' %('penny','penny'
))
# vs.
```

```
print('A {p} saved is a {p} earned.'.format(p='penn
y'))
```

```
A penny saved is a penny earned.
A penny saved is a penny earned.
```

Alignment, padding and precision with `.format()`

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

In [17]:

```
print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:9}'.format('Apples', 3.))
print('{0:8} | {1:9}'.format('Oranges', 10))
```

```
Fruit    | Quantity 
Apples   |       3.0
Oranges  |        10
```

By default, `.format()` aligns text to the left, numbers to the right. You can pass an optional `<`,`^`, or `>` to set a left, center or right alignment:

In [18]:

```
print('{0:<8} | {1:^8} | {2:>8}'.format('Left','Cent
er','Right'))
print('{0:<8} | {1:^8} | {2:>8}'.format(11,22,33))
```

```
Left     |  Center  |    Right
11       |    22    |       33
```

You can precede the aligment operator with a padding character

In [19]:

```
print('{0:=<8} | {1:-^8} | {2:.>8}'.format('Left','C
enter','Right'))
print('{0:=<8} | {1:-^8} | {2:.>8}'.format(11,22,33
))
```

```
Left==== | -Center- | ...Right
11====== | ---22--- | ......33
```

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

In [20]:

```
print('This is my ten-character, two-decimal number:
%10.2f' %13.579)
print('This is my ten-character, two-decimal number:
{0:10.2f}'.format(13.579))
```

```
This is my ten-character, two-decimal number:     1
3.58
This is my ten-character, two-decimal number:     1
3.58
```

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of ten characters.

For more information on the string `.format()` method visit https://docs.python.org/3/library/string.html#formatstrings (https://docs.python.org/3/library/string.html#formatstrings)

Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings offer several benefits over the older `.format()` string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through `.format(var)`.

In [21]:

```
name = 'Fred'

print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

Pass `!r` to get the string representation:

In [22]:

```
print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```

Float formatting follows `"result: {value:{width}.{precision}}"`

Where with the `.format()` method you might see `{value:10.4f}`, with f-strings this can become `{value:{10}.{6}}`

In [23]:

```
num = 23.45678
print("My 10 character, four decimal number is:{0:10.4f}".format(num))
print(f"My 10 character, four decimal number is:{num:{10}.{6}}")
```

```
My 10 character, four decimal number is:   23.4568
My 10 character, four decimal number is:   23.4568
```

Note that with f-strings, *precision* refers to the total number of digits, not just those following the decimal. This fits more closely with scientific notation and statistical analysis. Unfortunately, f-strings do not pad to the right of the decimal, even if precision allows it:

In [24]:

```
num = 23.45
```

```
print("My 10 character, four decimal number is:{0:1
0.4f}".format(num))
print(f"My 10 character, four decimal number is:{num
:{10}.{6}}")
```

```
My 10 character, four decimal number is:   23.4500
My 10 character, four decimal number is:     23.45
```

If this becomes important, you can always use `.format()` method syntax inside an f-string:

In [25]:

```
num = 23.45
print("My 10 character, four decimal number is:{0:1
0.4f}".format(num))
print(f"My 10 character, four decimal number is:{num
:10.4f}")
```

```
My 10 character, four decimal number is:   23.4500
My 10 character, four decimal number is:   23.4500
```

For more info on formatted string literals visit https://docs.python.org/3/reference/lexical_analysis.html#f-strings (https://docs.python.org/3/reference/lexical_analysis.html#f-strings)

That is the basics of string formatting!

Lists

Earlier when discussing strings we introduced the concept of a *sequence* in Python. Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

```
1.) Creating lists
2.) Indexing and Slicing Lists
3.) Basic List Methods
4.) Nesting Lists
5.) Introduction to List Comprehensions
```

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

In [1]:

```
# Assign a list to an variable named my_list
my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

```
In [2]:
```

```
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

```
In [3]:
```

```
len(my_list)
```

```
Out[3]:

4
```

Indexing and Slicing

Indexing and slicing work just like in strings. Let's make a new list to remind ourselves of how this works:

```
In [4]:
```

```
my_list = ['one','two','three',4,5]
```

```
In [5]:
```

```
# Grab element at index 0
my_list[0]
```

```
Out[5]:

'one'
```

```
In [6]:
```

```
# Grab index 1 and everything past it
my_list[1:]
```

```
Out[6]:

['two', 'three', 4, 5]
```

```
In [7]:
```

```
# Grab everything UP TO index 3
my_list[:3]
```

```
Out[7]:

['one', 'two', 'three']
```

We can also use + to concatenate lists, just like we did for strings.

```
In [8]:
```

```
my_list + ['new item']
```

```
Out[8]:

['one', 'two', 'three', 4, 5, 'new item']
```

Note: This doesn't actually change the original list!

In [9]:

```
my_list
```

Out[9]:

```
['one', 'two', 'three', 4, 5]
```

You would have to reassign the list to make the change permanent.

In [10]:

```
# Reassign
my_list = my_list + ['add new item permanently']
```

In [11]:

```
my_list
```

Out[11]:

```
['one', 'two', 'three', 4, 5, 'add new item permanen
tly']
```

We can also use the * for a duplication method similar to strings:

In [12]:

```
# Make the list double
my_list * 2
```

Out[12]:

```
['one',
 'two',
 'three',
 4,
 5,
 'add new item permanently',
 'one',
 'two',
 'three',
 4,
 5,
 'add new item permanently']
```

In [13]:

```
# Again doubling not permanent
my_list
```

Out[13]:

```
['one', 'two', 'three', 4, 5, 'add new item permanen
tly']
```

Basic List Methods

If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

Let's go ahead and explore some more special methods for lists:

In [14]:

```
# Create a new list
list1 = [1,2,3]
```

Use the **append** method to permanently add an item to the end of a list:

In [15]:

```
# Append
list1.append('append me!')
```

In [16]:

```
# Show
list1
```

Out[16]:

```
[1, 2, 3, 'append me!']
```

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

In [17]:

```
# Pop off the 0 indexed item
list1.pop(0)
```

Out[17]:

```
1
```

In [18]:

```
# Show
list1
```

Out[18]:

```
[2, 3, 'append me!']
```

In [19]:

```
# Assign the popped element, remember default popped
index is -1
popped_item = list1.pop()
```

In [20]:

```
popped_item
```

Out[20]:

```
'append me!'
```

In [21]:

```
# Show remaining list
list1
```

Out[21]:

```
[2, 3]
```

It should also be noted that lists indexing will return an error if there is no element at that index. For example:

In [22]:

```
list1[100]
```

```
----------------------------------------------------
-----------------------
IndexError                                Traceback
 (most recent call last)
<ipython-input-22-af6d2015fa1f> in <module>()
----> 1 list1[100]

IndexError: list index out of range
```

We can use the **sort** method and the **reverse** methods to also effect your lists:

In [23]:

```
new_list = ['a','e','x','b','c']
```

In [24]:

```
#Show
new_list
```

Out[24]:

```
['a', 'e', 'x', 'b', 'c']
```

In [25]:

```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

In [26]:

```
new_list
```

Out[26]:

```
['c', 'b', 'x', 'e', 'a']
```

In [27]:

```
# Use sort to sort the list (in this case alphabetic
al order, but for numbers it will go ascending)
new_list.sort()
```

In [28]:

```
new_list
```

Out[28]:

```
['a', 'b', 'c', 'e', 'x']
```

Nesting Lists

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

Let's see how this works!

In [29]:

```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]
```

In [30]:

```
# Show
matrix
```

Out[30]:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

In [31]:

```
# Grab first item in matrix object
matrix[0]
```

Out[31]:

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

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 2_Module.ipynb

naravishal Python Notes

1 contributor

553 lines (553 sloc) 10.8 KB

Comparison Operators

In this lecture we will be learning about Comparison Operators in Python. These operators will allow us to compare variables and output a Boolean value (True or False).

If you have any sort of background in Math, these operators should be very straight forward.

First we'll present a table of the comparison operators and then work through some examples:

Table of Comparison Operators

In the table below, a=3 and b=4.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Let's now work through quick examples of each of these.

Equal

```
In [1]: 2 == 2
Out[1]: True
```

```
In [2]: 1 == 0
Out[2]: False
```

Note that == is a *comparison* operator, while = is an *assignment* operator.

Not Equal

```
In [3]: 2 != 1
Out[3]: True
```

```
In [4]:  2 != 2
Out[4]:  False
```

Greater Than

```
In [5]:  2 > 1
Out[5]:  True
```

```
In [6]:  2 > 4
Out[6]:  False
```

Less Than

```
In [7]:  2 < 4
Out[7]:  True
```

```
In [8]:  2 < 1
Out[8]:  False
```

Greater Than or Equal to

```
In [9]:  2 >= 2
Out[9]:  True
```

```
In [10]:  2 >= 1
Out[10]:  True
```

Less than or Equal to

```
In [11]:  2 <= 2
Out[11]:  True
```

```
In [12]:  2 <= 4
Out[12]:  True
```

Great! Go over each comparison operator to make sure you understand what each one is saying. But hopefully this was straightforward for you.

Next we will cover chained comparison operators

Chained Comparison Operators

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

In this lecture we will learn how to chain comparison operators and we will also introduce two other important statements in Python: **and** and **or**.

Let's look at a few examples of using chains:

```
In [1]: 1 < 2 < 3

Out[1]: True
```

The above statement checks if 1 was less than 2 **and** if 2 was less than 3. We could have written this using an **and** statement in Python:

```
In [2]: 1<2 and 2<3

Out[2]: True
```

The **and** is used to make sure two checks have to be true in order for the total check to be true. Let's see another example:

```
In [3]: 1 < 3 > 2

Out[3]: True
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 3_Module_1.ipynb

naravishal Python Notes

1 contributor

608 lines (608 sloc) | 15.3 KB

Methods

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. Later on in the course we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.

Methods perform specific actions on an object and can also take arguments, just like a function. This lecture will serve as just a brief introduction to methods and get you thinking about overall design methods that we will touch back upon when we reach OOP in the course.

Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

You'll later see that we can think of methods as having an argument 'self' referring to the object itself. You can't see this argument but we will be using it later on in the course during the OOP lectures.

Let's take a quick look at what an example of the various methods a list has:

```
In [1]: # Create a simple list
        lst = [1,2,3,4,5]
```

Fortunately, with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. The methods for a list are:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

Let's try out a few of them:

append() allows us to add elements to the end of a list:

```
In [2]: lst.append(6)
```

```
In [3]: lst
Out[3]: [1, 2, 3, 4, 5, 6]
```

Great! Now how about count()? The count() method will count the number of occurrences of an element in a list.

```
In [4]: # Check how many times 2 shows up in the list
```

```
lst.count(2)
```

```
Out[4]: 1
```

You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function:

```
In [5]: help(lst.count)
```

```
Help on built-in function count:

count(...) method of builtins.list instance
    L.count(value) -> integer -- return number of occurrences of val
ue
```

Feel free to play around with the rest of the methods for a list. Later on in this section your quiz will involve using help and Google searching for methods of different types of objects!

Great! By this lecture you should feel comfortable calling methods of objects in Python!

Functions

Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

Function Topics

- def keyword
- simple example of a function
- calling a function with ()
- accepting parameters
- print versus return
- adding in logic inside a function
- multiple returns inside a function
- adding in loops inside a function

- tuple unpacking
- interactions between functions

def keyword

Let's see how to build out a function's syntax in Python. It has the following form:

```
In [1]: def name_of_function(arg1,arg2):
            '''
            This is where the function's Document String (docstring) goes.
            When you call help() on your function it will be printed out.
            '''
            # Do stuff here
            # Return desired result
```

We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (https://docs.python.org/3/library/functions.html) (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using Jupyter and Jupyter Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Simple example of a function

```
In [4]: def say_hello():
            print('hello')
```

Calling a function with ()

Call the function:

```
In [5]: say_hello()
```

```
hello
```

If you forget the parenthesis (), it will simply display the fact that say_hello is a function. Later on we

If you forget the parenthesis (), it will simply display the fact that say_hello is a function. Later on we will learn we can actually pass in functions into other functions! But for now, simply remember to call functions with ().

```
In [7]: say_hello
```

```
Out[7]: <function __main__.say_hello>
```

Accepting parameters (arguments)

Let's write a function that greets people with their name.

```
In [1]: def greeting(name):
            print(f'Hello {name}')
```

```
In [2]: greeting('Jose')
```

```
Hello Jose
```

Using return

So far we've only seen print() used, but if we actually want to save the resulting variable we need to use the **return** keyword.

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example: Addition function

```
In [6]: def add_num(num1,num2):
            return num1+num2
```

```
In [7]: add_num(4,5)
```

```
Out[7]: 9
```

```
In [8]: # Can also save as variable due to return
        result = add_num(4,5)
```

```
In [9]: print(result)
```

```
9
```

What happens if we input two strings?

```
In [10]: add_num('one','two')
```

```
Out[10]: 'onetwo'
```

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using `break`, `continue`, and `pass` statements in our code. We introduced these

Let's also start using `break`, `continue`, and `pass` statements in our code. We introduced these during the `while` lecture.

Check if a number is even

```
In [ ]:
```

Check if a number is even or odd

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 3_Module_2.ipynb

naravishal Python Notes

1 contributor

816 lines (816 sloc) | 18 KB

Lambda Expressions, Map, and Filter

Now its time to quickly learn about two built in functions, filter and map. Once we learn about how these operate, we can learn about the lambda expression, which will come in handy when you begin to develop your skills further!

map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

```
In [1]: def square(num):
            return num**2
```

```
In [2]: my_nums = [1,2,3,4,5]
```

```
In [5]: map(square,my_nums)

Out[5]: <map at 0x205baec21d0>
```

```
In [7]: # To get the results, either iterate through map()
        # or just cast to a list
        list(map(square,my_nums))

Out[7]: [1, 4, 9, 16, 25]
```

The functions can also be more complex

```
In [8]: def splicer(mystring):
            if len(mystring) % 2 == 0:
                return 'even'
            else:
                return mystring[0]
```

```
In [9]: mynames = ['John','Cindy','Sarah','Kelly','Mike']
```

```
In [10]: list(map(splicer,mynames))

Out[10]: ['even', 'C', 'S', 'K', 'even']
```

filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

```
In [12]: def check_even(num):
             return num % 2 == 0
```

```
In [13]: nums = [0,1,2,3,4,5,6,7,8,9,10]
```

```
In [15]: filter(check_even,nums)

Out[15]: <filter at 0x205baed4710>
```

```
In [16]: list(filter(check_even,nums))

Out[16]: [0, 2, 4, 6, 8, 10]
```

lambda expression

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

lambda's body is a single expression, not a block of statements.

- The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

Lets slowly break down a lambda expression by deconstructing a function:

```
In [17]: def square(num):
             result = num**2
             return result
```

```
In [18]: square(2)

Out[18]: 4
```

We could simplify it:

```
In [19]: def square(num):
             return num**2
```

```
In [20]: square(2)

Out[20]: 4
```

We could actually even write this all on one line.

```
In [21]: def square(num): return num**2
```

```
In [22]: square(2)

Out[22]: 4
```

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

```
In [23]: lambda num: num ** 2
```

```
Out[23]: <function __main__.<lambda>>
```

```
In [25]: # You wouldn't usually assign a name to a lambda expression, this is
         just for demonstration!
         square = lambda num: num **2
```

```
In [26]: square(2)
```

```
Out[26]: 4
```

So why would use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression. Let's repeat some of the examples from above with a lambda expression

```
In [29]: list(map(lambda num: num ** 2, my_nums))
```

```
Out[29]: [1, 4, 9, 16, 25]
```

```
In [30]: list(filter(lambda n: n % 2 == 0,nums))
```

```
Out[30]: [0, 2, 4, 6, 8, 10]
```

Here are a few more examples, keep in mind the more comples a function is, the harder it is to translate into a lambda expression, meaning sometimes its just easier (and often the only way) to create the def keyword function.

Lambda expression for grabbing the first character of a string:

```
In [31]: lambda s: s[0]
```

```
Out[31]: <function __main__.<lambda>>
```

Lambda expression for reversing a string:

```
In [32]: lambda s: s[::-1]
```

```
Out[32]: <function __main__.<lambda>>
```

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

```
In [34]: lambda x,y : x + y
```

```
Out[34]: <function __main__.<lambda>>
```

You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

the pandas library for data analysis works very well with lambda expressions.

*args and **kwargs

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

```
In [1]: def myfunc(a,b):
            return sum((a,b))*.05

        myfunc(40,60)

Out[1]: 5.0
```

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a** because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

```
In [2]: def myfunc(a=0,b=0,c=0,d=0,e=0):
            return sum((a,b,c,d,e))*.05

        myfunc(40,60,20)

Out[2]: 6.0
```

Obviously this is not a very efficient solution, and that's where `*args` comes in.

*args

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

```
In [3]: def myfunc(*args):
            return sum(args)*.05

        myfunc(40,60,20)

Out[3]: 6.0
```

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

```
In [4]: def myfunc(*spam):
            return sum(spam)*.05

        myfunc(40,60,20)

Out[4]: 6.0
```

```
Out[4]: 6.0
```

**kwargs

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

```
In [5]: def myfunc(**kwargs):
            if 'fruit' in kwargs:
                print(f"My favorite fruit is {kwargs['fruit']}")  # review S
        tring Formatting and f-strings if this syntax is unfamiliar
            else:
                print("I don't like fruit")

        myfunc(fruit='pineapple')
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 3_Module_3.ipynb

naravishal Python Notes

1 contributor

357 lines (357 sloc) | 10 KB

Nested Statements and Scope

Now that we have gone over writing our own functions, it's important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

Let's start with a quick thought experiment; imagine the following code:

In [1]:
```
x = 25

def printer():
    x = 50
    return x

# print(x)
# print(printer())
```

What do you imagine the output of printer() is? 25 or 50? What is the output of print x? 25 or 50?

In [2]:
```
print(x)
```
```
25
```

In [3]:
```
print(printer())
```
```
50
```

Interesting! But how does Python know which **x** you're referring to in your code? This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as **x** in this case) you are referencing in your code. Lets break down the rules:

This idea of scope in your code is very important to understand in order to properly assign and call variable names.

In simple terms, the idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.
2. Name references search (at most) four scopes, these are:
    - local
    - enclosing functions
    - global
    - built-in
3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

The statement in #2 above can be defined by the LEGB rule.

LEGB Rule:

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

Quick examples of LEGB

Local

In [4]:
```
# x is local here:
f = lambda x:x**2
```

Enclosing function locals

This occurs when we have a function inside a function (nested functions)

In [5]:
```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()

greet()
```
```
Hello Sammy
```

Note how Sammy was used, because the hello() function was enclosed inside of the greet function!

Global

Luckily in Jupyter a quick way to test for global variables is to see if another cell recognizes the variable!

In [6]:
```
print(name)
```
```
This is a global name
```

Built-in

These are the built-in function names in Python (don't overwrite these!)

In [7]:
```
len
```
Out[7]:
```
<function len>
```

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example:

In [8]:
```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

```
x is 50
Changed local x to 2
x is still 50
```

The first time that we print the value of the name **x** with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to **x**. The name **x** is local to our function. So, when we change the value of **x** in the function, the **x** defined in the main block remains unaffected.

With the last print statement, we display the value of **x** as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The `global` statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the `global` statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block.

Example:

In [9]:
```
x = 50

def func():
    global x
    print('This function is now using the global x!')
    print('Because of global x is: ', x)
    x = 2
    print('Ran func(), changed global x to', x)
print('Ran func(), changed global x to', x)
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 4_Module.ipynb

naravishal Python Notes

1 contributor

1306 lines (1306 sloc) 30.2 KB

Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the major obstacles for beginners when they are first starting to learn Python.

There are many, many tutorials and lessons covering OOP so feel free to Google search other lessons, and I have also put some links to other useful tutorials online at the bottom of this Notebook.

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

- Objects
- Using the *class* keyword
- Creating class attributes
- Creating methods in a class
- Learning about Inheritance
- Learning about Polymorphism
- Learning about Special Methods for classes

Lets start the lesson by remembering about the Basic Python Objects. For example:

```
In [1]: lst = [1,2,3]
```

Remember how we could call methods on a list?

```
In [2]: lst.count(2)
Out[2]: 1
```

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So let's explore Objects in general:

Objects

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

```
In [3]: print(type(1))
        print(type([]))
        print(type(()))
        print(type({}))

        <class 'int'>
        <class 'list'>
        <class 'tuple'>
        <class 'dict'>
```

So we know all these things are objects, so how can we create our own Object types? That is where the `class` keyword comes in.

class

User defined objects are created using the `class` keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object `lst` which was an instance of a list object.

Let see how we can use `class`:

In [4]:
```
# Create a new object type called Sample
class Sample:
    pass

# Instance of Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

By convention we give classes a name that starts with a capital letter. Note how `x` is now the reference to our new instance of a Sample class. In other words, we **instantiate** the Sample class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example.

Attributes

The syntax for creating an attribute is:

```
self.attribute = something
```

There is a special method called:

```
__init__()
```

This method is used to initialize the attributes of an object. For example:

In [5]:
```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Lets break down what we have above.The special method

```
__init__()
```

is called automatically right after the object has been created:

```
def __init__(self, breed):
```

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

```
 self.breed = breed
```

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

```
In [6]: sam.breed
Out[6]: 'Lab'
```

```
In [7]: frank.breed
Out[7]: 'Huskie'
```

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

In Python there are also *class object attributes*. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute *species* for the Dog class. Dogs, regardless of their breed, name, or other attributes, will always be mammals. We apply this logic in the following manner:

```
In [8]: class Dog:

            # Class Object Attribute
            species = 'mammal'

            def __init__(self,breed,name):
                self.breed = breed
                self.name = name
```

```
In [9]: sam = Dog('Lab','Sam')
```

```
In [10]: sam.name
Out[10]: 'Sam'
```

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

```
In [11]: sam.species
Out[11]: 'mammal'
```

Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

In [12]:
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

In the __init__ method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead.
In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi.

Now let's change the radius and see how that affects our Circle object:

In [13]:
```
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method.

Inheritance

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Let's see an example by incorporating our previous work on the Dog class:

In [14]:
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

In [15]:
```
d = Dog()
```
```
Animal created
Dog created
```

In [16]:
```
d.whoAmI()
```
```
Dog
```

In [17]:
```
d.eat()
```
```
Eating
```

In [18]:
```
d.bark()
```
```
Woof!
```

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

- It is shown by the eat() method.

The derived class modifies existing behavior of the base class.

- shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

method.

Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in. The best way to explain this is by example:

In [19]:
```python
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Here we have a Dog class and a Cat class, and each has a `.speak()` method. When called, each object's `.speak()` method returns a result unique to the object.

There a few different ways to demonstrate polymorphism. First, with a for loop:

In [20]:
```python
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Another is with functions:

In [21]:
```python
def pet_speak(pet):
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

A more common practice is to use abstract classes and inheritance. An abstract class is one that

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

In [22]:
```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by conv
ention only
        raise NotImplementedError("Subclass must implement abstract
 method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Real life examples of polymorphism include:

- opening different file types - different tools are needed to display Word, pdf and Excel files
- adding different objects - the + operator performs arithmetic and concatenation

Special Methods

Finally let's go over special methods. Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. For example let's create a Book class:

In [23]:
```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self
.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```

In [24]:
```
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

The `__init__(), __str__(), __len__() and __del__() methods`

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

Great! After this lecture you should have a basic understanding of how to create your own objects with class in Python. You will be utilizing this heavily in your next milestone project!

For more great resources on this topic, check out:

Jeff Knupp's Post (https://jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/)

Mozilla's Post (https://developer.mozilla.org/en-US/Learn/Python/Quickly_Learn_Object_Oriented_Programming)

Tutorial's Point (http://www.tutorialspoint.com/python/python_classes_objects.htm)

Official Documentation (https://docs.python.org/3/tutorial/classes.html)

Object Oriented Programming

Homework Assignment

Problem 1

Fill in the Line class methods to accept coordinates as a pair of tuples and return the slope and distance of the line.

In [1]:
```
class Line:

    def __init__(self,coor1,coor2):
        pass

    def distance(self):
        pass

    def slope(self):
        pass
```

```
In [2]: # EXAMPLE OUTPUT

        coordinate1 = (3,2)
        coordinate2 = (8,10)

        li = Line(coordinate1,coordinate2)
```

```
In [3]: li.distance()
```

```
Out[3]: 9.433981132056603
```

```
In [4]: li.slope()
```

```
Out[4]: 1.6
```

Problem 2

Fill in the class

```
In [5]: class Cylinder:

            def __init__(self,height=1,radius=1):
                pass

            def volume(self):
                pass

            def surface_area(self):
                pass
```

```
In [6]: # EXAMPLE OUTPUT
        c = Cylinder(2,3)
```

```
In [7]: c.volume()
```

```
Out[7]: 56.52
```

```
In [8]: c.surface_area()
```

```
Out[8]: 94.2
```

Object Oriented Programming

Homework Assignment

Problem 1

Fill in the Line class methods to accept coordinates as a pair of tuples and return the slope and distance of the line.

```
In [1]: class Line(object):
    def __init__(self,coor1,coor2):
        self.coor1 = coor1
        self.coor2 = coor2

    def distance(self):
        x1,y1 = self.coor1
        x2,y2 = self.coor2
        return ((x2-x1)**2 + (y2-y1)**2)**0.5

    def slope(self):
        x1,y1 = self.coor1
        x2,y2 = self.coor2
        return (y2-y1)/(x2-x1)
```

In [2]:
```
coordinate1 = (3,2)
coordinate2 = (8,10)

li = Line(coordinate1,coordinate2)
```

In [3]:
```
li.distance()
```

Out[3]:
```
9.433981132056603
```

In [4]:
```
li.slope()
```

Out[4]:
```
1.6
```

Problem 2

Fill in the class

In [5]:
```
class Cylinder:

    def __init__(self,height=1,radius=1):
        self.height = height
        self.radius = radius

    def volume(self):
        return self.height*3.14*(self.radius)**2

    def surface_area(self):
        top = 3.14 * (self.radius)**2
        return (2*top) + (2*3.14*self.radius*self.height)
```

In [6]:
```
c = Cylinder(2,3)
```

In [7]:
```
c.volume()
```

Out[7]:
```
56.52
```

In [8]:
```
c.surface_area()
```

Out[8]:
```
94.2
```

Object Oriented Programming Challenge - Solution

For this challenge, create a bank account class that has two attributes:

- owner
- balance

and two methods:

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 4_Module_project.ipynb

naravishal Python Notes

1 contributor

485 lines (485 sloc) | 12.2 KB

Milestone Project 1: Full Walk-through Code Solution

Below is the filled in code that goes along with the complete walk-through video. Check out the corresponding lecture videos for more information on this code!

Step 1: Write a function that can print out a board. Set up your board as a list, where each index 1-9 corresponds with a number on a number pad, so you get a 3 by 3 board representation.

In [1]:
```
from IPython.display import clear_output

def display_board(board):
    clear_output()  # Remember, this only works in jupyter!

    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')
```

TEST Step 1: run your function on a test version of the board list, and make adjustments as necessary

In [2]:
```
test_board = ['#','X','O','X','O','X','O','X','O','X']
display_board(test_board)
```

```
   |   |
 X | O | X
   |   |
-----------
   |   |
 O | X | O
   |   |
-----------
   |   |
 X | O | X
   |   |
```

Step 2: Write a function that can take in a player input and assign their marker as 'X' or 'O'. Think about using *while* loops to continually ask until you get a correct answer.

In [3]:
```
def player_input():
    marker = ''

    while not (marker == 'X' or marker == 'O'):
        marker = input('Player 1: Do you want to be X or O? ').upper
()
```

```
    if marker == 'X':
        return ('X', 'O')
    else:
        return ('O', 'X')
```

TEST Step 2: run the function to make sure it returns the desired output

In [4]:
```
player_input()
```

```
Player 1: Do you want to be X or O? X
```

Out[4]:
```
('X', 'O')
```

Step 3: Write a function that takes in the board list object, a marker ('X' or 'O'), and a desired position (number 1-9) and assigns it to the board.

In [5]:
```
def place_marker(board, marker, position):
    board[position] = marker
```

TEST Step 3: run the place marker function using test parameters and display the modified board

In [6]:
```
place_marker(test_board,'$',8)
display_board(test_board)
```

```
   |   |
 X | $ | X
   |   |
-----------
   |   |
 O | X | O
   |   |
-----------
   |   |
 X | O | X
   |   |
```

Step 4: Write a function that takes in a board and checks to see if someone has won.

In [7]:
```
def win_check(board,mark):

    return ((board[7] == mark and board[8] == mark and board[9] == m
ark) or # across the top
    (board[4] == mark and board[5] == mark and board[6] == mark) or
# across the middle
    (board[1] == mark and board[2] == mark and board[3] == mark) or
# across the bottom
    (board[7] == mark and board[4] == mark and board[1] == mark) or
# down the middle
    (board[8] == mark and board[5] == mark and board[2] == mark) or
# down the middle
    (board[9] == mark and board[6] == mark and board[3] == mark) or
# down the right side
    (board[7] == mark and board[5] == mark and board[3] == mark) or
# diagonal
    (board[9] == mark and board[5] == mark and board[1] == mark)) #
 diagonal
```

TEST Step 4: run the win_check function against our test_board - it should return True

```
In [8]: win_check(test_board,'X')
```

```
Out[8]: True
```

Step 5: Write a function that uses the random module to randomly decide which player goes first. You may want to lookup random.randint() Return a string of which player went first.

```
In [9]: import random

        def choose_first():
            if random.randint(0, 1) == 0:
                return 'Player 2'
            else:
                return 'Player 1'
```

Step 6: Write a function that returns a boolean indicating whether a space on the board is freely available.

```
In [10]: def space_check(board, position):

             return board[position] == ' '
```

Step 7: Write a function that checks if the board is full and returns a boolean value. True if full, False otherwise.

```
In [11]: def full_board_check(board):
             for i in range(1,10):
                 if space_check(board, i):
                     return False
             return True
```

Step 8: Write a function that asks for a player's next position (as a number 1-9) and then uses the function from step 6 to check if its a free position. If it is, then return the position for later use.

```
In [12]: def player_choice(board):
             position = 0

             while position not in [1,2,3,4,5,6,7,8,9] or not space_check(boa
         rd, position):
                 position = int(input('Choose your next position: (1-9) '))

             return position
```

Step 9: Write a function that asks the player if they want to play again and returns a boolean True if they do want to play again.

```
In [13]: def replay():

             return input('Do you want to play again? Enter Yes or No: ').low
         er().startswith('y')
er().startswith('y')
```

Step 10: Here comes the hard part! Use while loops and the functions you've made to run the game!

```
In [14]: print('Welcome to Tic Tac Toe!')

         while True:
             # Reset the board
             theBoard = [' '] * 10
             player1_marker, player2_marker = player_input()
             turn = choose_first()
             print(turn + ' will go first.')

             play_game = input('Are you ready to play? Enter Yes or No.')

             if play_game.lower()[0] == 'y':
                 game_on = True
             else:
                 game_on = False

             while game_on:
                 if turn == 'Player 1':
                     # Player1's turn.

                     display_board(theBoard)
                     position = player_choice(theBoard)
                     place_marker(theBoard, player1_marker, position)

                     if win_check(theBoard, player1_marker):
                         display_board(theBoard)
                         print('Congratulations! You have won the game!')
                         game_on = False
                     else:
                         if full_board_check(theBoard):
                             display_board(theBoard)
                             print('The game is a draw!')
                             break
                         else:
                             turn = 'Player 2'

                 else:
                     # Player2's turn.
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 5_Module.ipynb

naravishal Python Notes

1 contributor

1230 lines (1230 sloc) | 37.2 KB

Errors and Exception Handling

In this lecture we will learn about Errors and Exception Handling in Python. You've definitely already encountered errors by this point in the course. For example:

In [1]:
```
print('Hello)
```

```
  File "<ipython-input-1-db8c9988558c>", line 1
    print('Hello)
                 ^
SyntaxError: EOL while scanning string literal
```

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

You can check out the full list of built-in exceptions here (https://docs.python.org/3/library/exceptions.html). Now let's learn how to handle errors and exceptions in our own code.

try and except

The basic terminology and syntax used to handle errors in Python are the `try` and `except` statements. The code which can cause an exception to occur is put in the `try` block and the handling of the exception is then implemented in the `except` block of code. The syntax follows:

```
try:
   You do your operations here...
   ...
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ...
else:
   If there is no exception then execute this block.
```

We can also just check for any exception with just using `except:` To get a better understanding of all this let's check out an example: We will look at some code that opens and writes a file:

In [2]:
```
try:
    f = open('testfile','w')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute
this print statement
```

```
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Content written successfully
```

Now let's see what would happen if we did not have write permission (opening only with 'r'):

In [3]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this print statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Error: Could not find file or read data
```

Great! Notice how we only printed a statement! The code still ran and we were able to continue doing actions and running code blocks. This is extremely useful when you have to account for possible input errors in your code. You can be prepared for the error and keep running code, instead of your code just breaking as we saw above.

We could have also just said `except:` if we weren't sure what exception would occur. For example:

In [4]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except:
    # This will check for any exception and then execute this print statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Error: Could not find file or read data
```

Great! Now we don't actually need to memorize that list of exception types! Now what if we kept wanting to run code after the exception occurred? This is where `finally` comes in.

finally

The `finally:` block of code will always be run regardless if there was an exception in the `try` code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

For example:

```
In [5]: try:
            f = open("testfile", "w")
            f.write("Test write statement")
            f.close()
        finally:
            print("Always execute finally code blocks")
```

```
Always execute finally code blocks
```

We can use this in conjunction with `except`. Let's see a new example that will take into account a user providing the wrong input:

```
In [6]: def askint():
            try:
                val = int(input("Please enter an integer: "))
            except:
                print("Looks like you did not enter an integer!")

            finally:
                print("Finally, I executed!")
            print(val)
```

```
In [7]: askint()
```

```
Please enter an integer: 5
Finally, I executed!
5
```

```
In [8]: askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
--------------------------------------------------------------------
-------
UnboundLocalError                         Traceback (most recent cal
l last)
<ipython-input-8-cc291aa76c10> in <module>()
----> 1 askint()

<ipython-input-6-c97dd1c75d24> in askint()
      7     finally:
      8         print("Finally, I executed!")
----> 9     print(val)

UnboundLocalError: local variable 'val' referenced before assignment
```

Notice how we got an error when trying to print val (because it was never properly assigned). Let's remedy this by asking the user and checking to make sure the input type is an integer:

```
In [9]: def askint():
            try:
                val = int(input("Please enter an integer: "))
            except:
                print("Looks like you did not enter an integer!")
```

```
        val = int(input("Try again-Please enter an integer: "))
    finally:
        print("Finally, I executed!")
    print(val)
```

In [10]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Try again-Please enter an integer: four
Finally, I executed!
--------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-9-92b5f751eb01> in askint()
      2     try:
----> 3         val = int(input("Please enter an integer: "))
      4     except:

ValueError: invalid literal for int() with base 10: 'five'

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent cal
l last)
<ipython-input-10-cc291aa76c10> in <module>()
----> 1 askint()

<ipython-input-9-92b5f751eb01> in askint()
      4     except:
      5         print("Looks like you did not enter an integer!")
----> 6         val = int(input("Try again-Please enter an integer:
 "))
      7     finally:
      8         print("Finally, I executed!")

ValueError: invalid literal for int() with base 10: 'four'
```

Hmmm...that only did one check. How can we continually keep checking? We can use a while loop!

In [11]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            break
        finally:
            print("Finally, I executed!")
        print(val)
```

In [12]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: four
Please enter an integer: four
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 3
Yep that's an integer!
Finally, I executed!
```

So why did our function print "Finally, I executed!" after each trial, yet it never printed `val` itself? This is because with a try/except/finally clause, any `continue` or `break` statements are reserved until *after* the try clause is completed. This means that even though a successful input of **3** brought us to the `else:` block, and a `break` statement was thrown, the try clause continued through to `finally:` before breaking out of the while loop. And since `print(val)` was outside the try clause, the `break` statement prevented it from running.

Let's make one final adjustment:

In [13]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            print(val)
            break
        finally:
            print("Finally, I executed!")
```

In [14]:
```
askint()
```

```
Please enter an integer: six
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 6
Yep that's an integer!
6
Finally, I executed!
```

Great! Now you know how to handle errors and exceptions in Python with the try, except, else, and finally notation!

Errors and Exceptions Homework

Problem 1

Handle the exception thrown by the code below by using `try` and `except` blocks.

In [1]:
```
for i in ['a','b','c']:
    print(i**2)
```

```
--------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-1-c35f41ad7311> in <module>()
      1 for i in ['a','b','c']:
----> 2     print(i**2)

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'i
nt'
```

Problem 2

Handle the exception thrown by the code below by using try and except blocks. Then use a finally block to print 'All Done.'

In [2]:
```
x = 5
y = 0

z = x/y
```

```
--------------------------------------------------------------------
-------
ZeroDivisionError                         Traceback (most recent cal
l last)
<ipython-input-2-6f985c4c80dd> in <module>()
      2 y = 0
      3
----> 4 z = x/y

ZeroDivisionError: division by zero
```

Problem 3

Write a function that asks for an integer and prints the square of it. Use a while loop with a try, except, else block to account for incorrect inputs.

In [3]:
```
def ask():
    pass
```

In [4]:
```
ask()
```

```
Input an integer: null
An error occurred! Please try again!
Input an integer: 2
Thank you, your number squared is:  4
```

Great Job!

Errors and Exceptions Homework - Solution

Problem 1

Handle the exception thrown by the code below by using try and except blocks.

In [1]:
```
try:
    for i in ['a','b','c']:
        print(i**2)
except:
    print("An error occurred!")
```

```
An error occurred!
```

Problem 2

Handle the exception thrown by the code below by using `try` and `except` blocks. Then use a `finally` block to print 'All Done.'

In [2]:
```
x = 5
y = 0
try:
    z = x/y
except ZeroDivisionError:
    print("Can't divide by Zero!")
finally:
    print('All Done!')
```

```
Can't divide by Zero!
All Done!
```

Problem 3

Write a function that asks for an integer and prints the square of it. Use a `while` loop with a `try`, `except`, `else` block to account for incorrect inputs.

In [3]:
```
def ask():

    while True:
        try:
            n = int(input('Input an integer: '))
        except:
            print('An error occurred! Please try again!')
            continue
        else:
            break


    print('Thank you, your number squared is: ',n**2)
```

In [4]:
```
ask()
```

```
Input an integer: null
An error occurred! Please try again!
Input an integer: 2
Thank you, your number squared is:  4
```

Great Job!

Unit Testing

Unit Testing

Equally important as writing good code is writing good tests. Better to find bugs yourself than have them reported to you by end users!

For this section we'll be working with files outside the notebook. We'll save our code to a .py file, and then save our test script to another .py file. Normally we would code these files using a text editor like Brackets or Atom, or inside an IDE like Spyder or Pycharm. But, since we're here, let's use Jupyter!

Recall that with some IPython magic we can write the contents of a cell to a file using `%%writefile`.
Something we haven't seen yet; you can run terminal commands from a jupyter cell using `!`

Testing tools

There are dozens of good testing libraries out there. Most are third-party packages that require an install, such as:

- pylint (https://www.pylint.org/)
- pyflakes (https://pypi.python.org/pypi/pyflakes/)
- pep8 (https://pypi.python.org/pypi/pep8)

These are simple tools that merely look at your code, and they'll tell you if there are style issues or simple problems like variable names being called before assignment.

A far better way to test your code is to write tests that send sample data to your program, and compare what's returned to a desired outcome.
Two such tools are available from the standard library:

- unittest (https://docs.python.org/3/library/unittest.html)
- doctest (https://docs.python.org/3/library/doctest.html)

Let's look at pylint first, then we'll do some heavier lifting with unittest.

`pylint`

`pylint` tests for style as well as some very basic program logic.

First, if you don't have it already (and you probably do, as it's part of the Anaconda distribution), you should install `pylint`.
Once that's done feel free to comment out the cell, you won't need it anymore.

```
In [ ]: ! pip install pylint
```

Let's save a very simple script:

```
In [1]: %%writefile simple1.py
        a = 1
        b = 2
        print(a)
        print(B)
```

```
Overwriting simple1.py
```

Now let's check it using pylint

In [2]:
```
! pylint simple1.py
```

```
************* Module simple1
C:  4, 0: Final newline missing (missing-final-newline)
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Invalid constant name "a" (invalid-name)
C:  2, 0: Invalid constant name "b" (invalid-name)
E:  4, 6: Undefined variable 'B' (undefined-variable)

--------------------------------------------------------------------
-

Your code has been rated at -12.50/10 (previous run: 8.33/10, -20.8
3)



No config file found, using default configuration
```

Pylint first lists some styling issues - it would like to see an extra newline at the end, modules and function definitions should have descriptive docstrings, and single characters are a poor choice for variable names.

More importantly, however, pylint identified an error in the program - a variable called before assignment. This needs fixing.

Note that pylint scored our program a negative 12.5 out of 10. Let's try to improve that!

In [3]:
```
%%writefile simple1.py
"""
A very simple script.
"""

def myfunc():
    """
    An extremely simple function.
    """
    first = 1
    second = 2
    print(first)
    print(second)

myfunc()
```

```
Overwriting simple1.py
```

In [4]:
```
! pylint simple1.py
```

```
************* Module simple1
C: 14, 0: Final newline missing (missing-final-newline)

--------------------------------------------------------------------
-

Your code has been rated at 8.33/10 (previous run: -12.50/10, +20.8
3)
```

```
No config file found, using default configuration
```

Much better! Our score climbed to 8.33 out of 10. Unfortunately, the final newline has to do with how jupyter writes to a file, and there's not much we can do about that here. Still, pylint helped us troubleshoot some of our problems. But what if the problem was more complex?

In [5]:
```
%%writefile simple2.py
"""
A very simple script.
"""

def myfunc():
    """
    An extremely simple function.
    """
    first = 1
    second = 2
    print(first)
    print('second')

myfunc()
```

```
Overwriting simple2.py
```

In [6]:
```
! pylint simple2.py
```

```
************* Module simple2
C: 14, 0: Final newline missing (missing-final-newline)
W: 10, 4: Unused variable 'second' (unused-variable)

------------------------------------------------------------------

Your code has been rated at 6.67/10 (previous run: 6.67/10, +0.00)



No config file found, using default configuration
```

pylint tells us there's an unused variable in line 10, but it doesn't know that we might get an unexpected output from line 12! For this we need a more robust set of tools. That's where `unittest` comes in.

unittest

`unittest` lets you write your own test programs. The goal is to send a specific set of data to your program, and analyze the returned results against an expected result.

Let's generate a simple script that capitalizes words in a given string. We'll call it **cap.py**.

In [7]:
```
%%writefile cap.py
def cap_text(text):
    return text.capitalize()
```

```
Overwriting cap.py
```

Now we'll write a test script. We can call it whatever we want, but **test_cap.py** seems an obvious choice.

When writing test functions, it's best to go from simple to complex, as each function will be run in order. Here we'll test simple, one-word strings, followed by a test of multiple word strings.

In [8]:
```
%%writefile test_cap.py
import unittest
import cap

class TestCap(unittest.TestCase):

    def test_one_word(self):
        text = 'python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Python')

    def test_multiple_words(self):
        text = 'monty python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Monty Python')

if __name__ == '__main__':
    unittest.main()
```

```
Overwriting test_cap.py
```

In [9]:
```
! python test_cap.py
```

```
F.
====================================================================
==
FAIL: test_multiple_words (__main__.TestCap)
--------------------------------------------------------------------
--
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 6_Module.ipynb

naravishal Python Notes

1 contributor

425 lines (425 sloc) | 11 KB

Iterators and Generators

In this section of the course we will be learning the difference between iteration and generation in Python and how to construct our own Generators with the *yield* statement. Generators allow us to generate as we go along, instead of holding everything in memory.

We've touched on this topic in the past when discussing certain built-in Python functions like **range()**, **map()** and **filter()**.

Let's explore a little deeper. We've learned how to create functions with `def` and the `return` statement. Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off. This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a `yield` statement.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as *state suspension*.

To start getting a better understanding of generators, let's go ahead and see how we can create some.

In [1]:
```python
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

In [2]:
```python
for x in gencubes(10):
    print(x)
```

```
0
1
8
27
64
125
216
343
512
729
```

Great! Now since we have a generator function we don't have to keep track of every single cube we created.

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don’t want to allocate the memory for all of the results at the same time.

Let's create another example generator which calculates fibonacci (https://en.wikipedia.org/wiki/Fibonacci_number) numbers:

In [3]:
```python
def genfibon(n):
    """
    Generate a fibonnaci sequence up to n
    """
    a = 1
    b = 1
    for i in range(n):
        yield a
        a,b = b,a+b
```

In [4]:
```python
for num in genfibon(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

What if this was a normal function, what would it look like?

In [5]:
```python
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

In [6]:
```python
fibon(10)
```

Out[6]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Notice that if we call some huge value of n (like 100000) the second function will have to keep track of every single result, when in our case we actually only care about the previous result to generate the next one!

next() and iter() built-in functions

A key to fully understanding generators is the next() function and the iter() function.

The next() function allows us to access the next element in a sequence. Lets check it out:

In [7]:
```python
def simple_gen():
    for x in range(3):
        yield x
```

```
In [8]: # Assign simple_gen
        g = simple_gen()
```

```
In [9]: print(next(g))
```

```
0
```

```
In [10]: print(next(g))
```

```
1
```

```
In [11]: print(next(g))
```

```
2
```

```
In [12]: print(next(g))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-12-1dfb29d6357e> in <module>()
----> 1 print(next(g))

StopIteration:
```

After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.

You might be wondering that why don’t we get this error while using a for loop? A for loop automatically catches this error and stops calling next().

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

```
In [13]: s = 'hello'

         #Iterate over string
         for let in s:
             print(let)
```

```
h
e
l
l
o
```

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

```
In [14]: next(s)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-61c30b5fe1d5> in <module>()
----> 1 next(s)
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 6_Module_Project.ipynb

naravishal Python Notes

1 contributor

321 lines (321 sloc) | 10.1 KB

Milestone Project 2 - Solution Code

Below is an implementation of a simple game of Blackjack. Notice the use of OOP and classes for the cards and hands.

In [3]:
```
# IMPORT STATEMENTS AND VARIABLE DECLARATIONS:

import random

suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs')
ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'N
ine', 'Ten', 'Jack', 'Queen', 'King', 'Ace')
values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7
, 'Eight':8,
            'Nine':9, 'Ten':10, 'Jack':10, 'Queen':10, 'King':10, 'A
ce':11}

playing = True

# CLASS DEFINTIONS:

class Card:

    def __init__(self,suit,rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return self.rank + ' of ' + self.suit


class Deck:

    def __init__(self):
        self.deck = []  # start with an empty list
        for suit in suits:
            for rank in ranks:
                self.deck.append(Card(suit,rank))

    def __str__(self):
        deck_comp = ''  # start with an empty string
        for card in self.deck:
            deck_comp += '\n '+card.__str__() # add each Card objec
t's print string
        return 'The deck has:' + deck_comp

    def shuffle(self):
        random.shuffle(self.deck)

    def deal(self):
        single_card = self.deck.pop()
        return single_card


class Hand:

    def __init__(self):
        self.cards = []  # start with an empty list as we did in the
```

```
Deck class
        self.value = 0   # start with zero value
        self.aces = 0    # add an attribute to keep track of aces

    def add_card(self,card):
        self.cards.append(card)
        self.value += values[card.rank]
        if card.rank == 'Ace':
            self.aces += 1  # add to self.aces

    def adjust_for_ace(self):
        while self.value > 21 and self.aces:
            self.value -= 10
            self.aces -= 1


class Chips:

    def __init__(self):
        self.total = 100
        self.bet = 0

    def win_bet(self):
        self.total += self.bet

    def lose_bet(self):
        self.total -= self.bet


# FUNCTION DEFINITIONS:

def take_bet(chips):

    while True:
        try:
            chips.bet = int(input('How many chips would you like to
 bet? '))
        except ValueError:
            print('Sorry, a bet must be an integer!')
        else:
            if chips.bet > chips.total:
                print("Sorry, your bet can't exceed",chips.total)
            else:
                break

def hit(deck,hand):
    hand.add_card(deck.deal())
    hand.adjust_for_ace()

def hit_or_stand(deck,hand):
    global playing

    while True:
        x = input("Would you like to Hit or Stand? Enter 'h' or 's'
 ")

        if x[0].lower() == 'h':
            hit(deck,hand)  # hit() function defined above

        elif x[0].lower() == 's':
            print("Player stands. Dealer is playing.")
            playing = False

        else:
            print("Sorry, please try again.")
            continue
        break


def show_some(player,dealer):
    print("\nDealer's Hand:")
    print(" <card hidden>")
    print('',dealer.cards[1])
    print("\nPlayer's Hand:", *player.cards, sep='\n ')

def show_all(player,dealer):
    print("\nDealer's Hand:", *dealer.cards, sep='\n ')
    print("Dealer's Hand =",dealer.value)
    print("\nPlayer's Hand:", *player.cards, sep='\n ')
    print("Player's Hand =",player.value)

def player_busts(player,dealer,chips):
    print("Player busts!")
    chips.lose_bet()

def player_wins(player,dealer,chips):
    print("Player wins!")
    chips.win_bet()

def dealer_busts(player,dealer,chips):
    print("Dealer busts!")
    chips.win_bet()

def dealer_wins(player,dealer,chips):
    print("Dealer wins!")
    chips.lose_bet()

def push(player,dealer):
    print("Dealer and Player tie! It's a push.")

# GAMEPLAY!

while True:
    print('Welcome to BlackJack! Get as close to 21 as you can witho
ut going over!\n\
    Dealer hits until she reaches 17. Aces count as 1 or 11.')

    # Create & shuffle the deck, deal two cards to each player
    deck = Deck()
    deck.shuffle()

    player_hand = Hand()
    player_hand.add_card(deck.deal())
    player_hand.add_card(deck.deal())

    dealer_hand = Hand()
    dealer_hand.add_card(deck.deal())
    dealer_hand.add_card(deck.deal())

    # Set up the Player's chips
    player_chips = Chips()  # remember the default value is 100

    # Prompt the Player for their bet:
```

```
    take_bet(player_chips)

    # Show the cards:
    show_some(player_hand,dealer_hand)

    while playing:  # recall this variable from our hit_or_stand fun
ction

        # Prompt for Player to Hit or Stand
        hit_or_stand(deck,player_hand)
        show_some(player_hand,dealer_hand)

        if player_hand.value > 21:
            player_busts(player_hand,dealer_hand,player_chips)
            break

    # If Player hasn't busted, play Dealer's hand
    if player_hand.value <= 21:

        while dealer_hand.value < 17:
            hit(deck,dealer_hand)

        # Show all cards
        show_all(player_hand,dealer_hand)

        # Test different winning scenarios
        if dealer_hand.value > 21:
            dealer_busts(player_hand,dealer_hand,player_chips)

        elif dealer_hand.value > player_hand.value:
            dealer_wins(player_hand,dealer_hand,player_chips)

        elif dealer_hand.value < player_hand.value:
            player_wins(player_hand,dealer_hand,player_chips)

        else:
            push(player_hand,dealer_hand)

    # Inform Player of their chips total
    print("\nPlayer's winnings stand at",player_chips.total)

    # Ask to play again
    new_game = input("Would you like to play another hand? Enter 'y'
or 'n' ")
    if new_game[0].lower()=='y':
        playing=True
        continue
    else:
        print("Thanks for playing!")
        break
```

```
Welcome to BlackJack! Get as close to 21 as you can without going ov
er!
    Dealer hits until she reaches 17. Aces count as 1 or 11.
How many chips would you like to bet? 50

Dealer's Hand:
 <card hidden>
 Seven of Diamonds

Player's Hand:
 Jack of Clubs
 Jack of Clubs
 Three of Diamonds
Would you like to Hit or Stand? Enter 'h' or 's' h

Dealer's Hand:
 <card hidden>
 Seven of Diamonds

Player's Hand:
 Jack of Clubs
 Three of Diamonds
 Six of Hearts
Would you like to Hit or Stand? Enter 'h' or 's' s
Player stands. Dealer is playing.

Dealer's Hand:
 <card hidden>
 Seven of Diamonds

Player's Hand:
 Jack of Clubs
 Three of Diamonds
 Six of Hearts

Dealer's Hand:
 Ace of Hearts
 Seven of Diamonds
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 7_Module.ipynb

naravishal Python Notes

1 contributor

958 lines (958 sloc) | 44.2 KB

map()

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

```
map(function, iterable, ...)
```

map() returns an *iterator* - that is, map() returns a special object that yields one result at a time as needed. We will learn more about iterators and generators in a future lecture. For now, since our examples are so small, we will cast map() as a list to see the results immediately.

When we went over list comprehensions we created a small expression to convert Celsius to Fahrenheit. Let's do the same here but use map:

```
In [1]: def fahrenheit(celsius):
            return (9/5)*celsius + 32

        temps = [0, 22.5, 40, 100]
```

Now let's see map() in action:

```
In [2]: F_temps = map(fahrenheit, temps)

        #Show
        list(F_temps)

Out[2]: [32.0, 72.5, 104.0, 212.0]
```

In the example above, map() applies the fahrenheit function to every item in temps. However, we don't have to define our functions beforehand; we can use a lambda expression instead:

```
In [3]: list(map(lambda x: (9/5)*x + 32, temps))

Out[3]: [32.0, 72.5, 104.0, 212.0]
```

Great! We got the same result! Using map with lambda expressions is much more common since the entire purpose of map() is to save effort on having to create manual for loops.

map() with multiple iterables

map() can accept more than one iterable. The iterables should be the same length - in the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

For instance, if our function is trying to add two values **x** and **y**, we can pass a list of **x** values and another list of **y** values to map(). The function (or lambda) will be fed the 0th index from each list, and then the 1st index, and so on until the n-th index is reached.

Let's see this in action with two and then three lists:

```
In [4]: a = [1,2,3,4]
        b = [5,6,7,8]
```

```
c = [9,10,11,12]

list(map(lambda x,y:x+y,a,b))
```

Out[4]: [6, 8, 10, 12]

In [5]:
```
# Now all three lists
list(map(lambda x,y,z:x+y+z,a,b,c))
```

Out[5]: [15, 18, 21, 24]

We can see in the example above that the parameter **x** gets its values from the list **a**, while **y** gets its values from **b** and **z** from list **c**. Go ahead and play with your own example to make sure you fully understand mapping to more than one iterable.

Great job! You should now have a basic understanding of the map() function.

reduce()

Many times students have difficulty understanding reduce() so pay careful attention to this lecture. The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [s1, s2, s3, ... , sn], calling reduce(function, sequence) works like this:

- At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
- The list on which reduce() works looks now like this: [function(s1, s2), s3, ... , sn]
- In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
- The list looks like this now: [function(function(s1, s2),s3), ... , sn]
- It continues like this until just one element is left and return this element as the result of reduce()

Let's see an example:

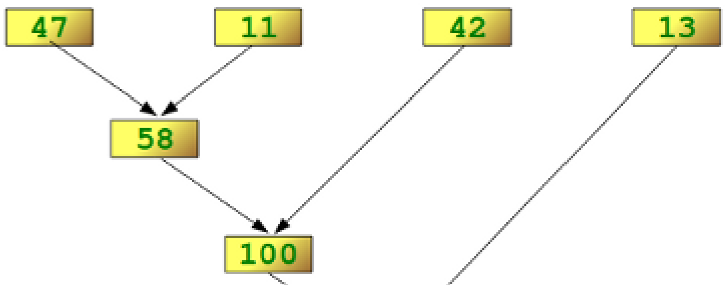
In [1]:
```
from functools import reduce

lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

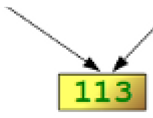
Out[1]: 113

Lets look at a diagram to get a better understanding of what is going on here:

In [2]:
```
from IPython.display import Image
Image('http://www.python-course.eu/images/reduce_diagram.png')
```

Out[2]:





Note how we keep reducing the sequence until a single final value is obtained. Lets see another example:

```
In [3]: #Find the maximum of a sequence (This already exists as max())
        max_find = lambda a,b: a if (a > b) else b
```

```
In [4]: #Find max
        reduce(max_find,lst)
```

```
Out[4]: 47
```

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!

filter

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

The function filter(function,list) needs a function as its first argument. The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns True will the element of the iterable be included in the result.

Like map(), filter() returns an *iterator* - that is, filter yields one result at a time as needed. Iterators and generators will be covered in an upcoming lecture. For now, since our examples are so small, we will cast filter() as a list to see our results immediately.

Let's see some examples:

```
In [1]: #First let's make a function
        def even_check(num):
            if num%2 ==0:
                return True
```

Now let's filter a list of numbers. Note: putting the function into filter without any parentheses might feel strange, but keep in mind that functions are objects as well.

```
In [2]: lst =range(20)

        list(filter(even_check,lst))
```

```
Out[2]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

filter() is more commonly used with lambda functions, because we usually use filter for a quick job where we don't want to write an entire function. Let's repeat the example above using a lambda expression:

```
In [3]: list(filter(lambda x: x%2==0,lst))
```

```
Out[3]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Great! You should now have a solid understanding of filter() and how to apply it to your code!

enumerate()

In this lecture we will learn about an extremely useful built-in function: enumerate(). Enumerate allows you to keep a count as you iterate through an object. It does this by returning a tuple in the form (count,element). The function itself is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

Example

```
In [1]: lst = ['a','b','c']

        for number,item in enumerate(lst):
            print(number)
            print(item)
```

```
0
a
1
b
2
c
```

enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker. For example:

```
In [2]: for count,item in enumerate(lst):
            if count >= 2:
                break
            else:
                print(item)
```

```
a
b
```

enumerate() takes an optional "start" argument to override the default value of zero:

```
In [3]: months = ['March','April','May','June']

        list(enumerate(months,start=3))
```

```
Out[3]: [(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

Great! You should now have a good understanding of enumerate and its potential use cases.

all() and any()

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. It is the same as this function code:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Let's see a few examples of these functions. They should be fairly straightforward:

```
In [1]: lst = [True,True,False,True]
```

```
In [2]: all(lst)
```

```
Out[2]: False
```

Returns False because not all elements are True.

```
In [3]: any(lst)
```

```
Out[3]: True
```

Returns True because at least one of the elements in the list is True

There you have it, you should have an understanding of how to use any() and all() in your code.

complex()

complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.

If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns

0j.

If you are doing math or engineering that requires complex numbers (such as dynamics, control systems, or impedance of a circuit) this is a useful tool to have in Python.

Let's see some examples:

```
In [1]: # Create 2+3j
        complex(2,3)

Out[1]: (2+3j)
```

```
In [2]: complex(10,1)

Out[2]: (10+1j)
```

We can also pass strings:

```
In [3]: complex('12+2j')

Out[3]: (12+2j)
```

That's really all there is to this useful function. Keep it in mind if you are ever dealing with complex numbers in Python!

Built-in Functions Test Solutions

For this test, you should use built-in functions and be able to write the requested functions in one line.

Problem 1

Use map() to create a function which finds the length of each word in the phrase (broken by spaces) and return the values in a list.

The function will have an input of a string, and output a list of integers.

```
In [1]: def word_lengths(phrase):

            return list(map(len, phrase.split()))
```

```
In [2]: word_lengths('How long are the words in this phrase')

Out[2]: [3, 4, 3, 3, 5, 2, 4, 6]
```

Problem 2

Use reduce() to take a list of digits and return the number that they correspond to. For example, [1,2,3] corresponds to one-hundred-twenty-three.
Do not convert the integers to strings!

```
In [3]: from functools import reduce
In [3]: from functools import reduce

        def digits_to_num(digits):

            return reduce(lambda x,y:x*10 + y,digits)
```

```
In [4]: digits_to_num([3,4,3,2,1])
```

```
Out[4]: 34321
```

Problem 3

Use filter() to return the words from a list of words which start with a target letter.

```
In [5]: def filter_words(word_list, letter):
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 8_Module.ipynb

naravishal Python Notes

1 contributor

648 lines (648 sloc) 15.4 KB

Decorators

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions. Make sure to run every cell in this Notebook for this lecture to look the same on your own computer.

So let's break down the steps:

Functions Review

```
In [1]: def func():
            return 1
```

```
In [2]: func()
```

```
Out[2]: 1
```

Scope Review

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

```
In [ ]: s = 'Global Variable'

        def check_for_locals():
            print(locals())
```

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the `locals()` and `globals()` functions. For example:

```
In [ ]: print(globals())
```

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

```
In [ ]: print(globals().keys())
```

Note how **s** is there, the Global Variable we defined as a string:

```
In [ ]: globals()['s']
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

```
In [ ]: check_for_locals()
```

Great! Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions. Lets start with some simple examples:

```
In [3]: def hello(name='Jose'):
            return 'Hello '+name
```

```
In [4]: hello()
```

```
Out[4]: 'Hello Jose'
```

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

```
In [5]: greet = hello
```

```
In [6]: greet
```

```
Out[6]: <function __main__.hello>
```

```
In [7]: greet()
```

```
Out[7]: 'Hello Jose'
```

So what happens when we delete the name **hello**?

```
In [8]: del hello
```

```
In [9]: hello()
```

```
--------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-9-a75d7781aaeb> in <module>()
----> 1 hello()

NameError: name 'hello' is not defined
```

```
In [10]: greet()
```

```
Out[10]: 'Hello Jose'
```

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

Functions within functions

Great! So we've seen how we can treat functions as objects, now let's see how we can define functions inside of other functions:

```
In [11]: def hello(name='Jose'):
    print('The hello() function has been executed')

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    print(greet())
    print(welcome())
    print("Now we are back inside the hello() function")
```

In [12]:
```
hello()
```

```
The hello() function has been executed
	 This is inside the greet() function
	 This is inside the welcome() function
Now we are back inside the hello() function
```

In [13]:
```
welcome()
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-13-a401d7101853> in <module>()
----> 1 welcome()

NameError: name 'welcome' is not defined
```

Note how due to scope, the welcome() function is not defined outside of the hello() function. Now lets learn about returning functions from within functions:

Returning Functions

In [14]:
```
def hello(name='Jose'):

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    if name == 'Jose':
        return greet
    else:
        return welcome
```

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

In [15]:
```
x = hello()
```

In [16]:
```
x
```

Out[16]:
```
<function __main__.hello.<locals>.greet>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

In [17]:
```
print(x())
```

```
        This is inside the greet() function
```

Let's take a quick look at the code again.

In the `if/else` clause we are returning `greet` and `welcome`, not `greet()` and `welcome()`.

This is because when you put a pair of parentheses after it, the function gets executed; whereas if you don’t put parentheses after it, then it can be passed around and can be assigned to other variables without executing it.

When we write `x = hello()`, hello() gets executed and because the name is Jose by default, the function `greet` is returned. If we change the statement to `x = hello(name = "Sam")` then the `welcome` function will be returned. We can also do `print(hello()())` which outputs *This is inside the greet() function*.

Functions as Arguments

Now let's see how we can pass functions as arguments into other functions:

In [18]:
```
def hello():
    return 'Hi Jose!'

def other(func):
    print('Other code would go here')
    print(func())
```

In [19]:
```
other(hello)
```

```
Other code would go here
Hi Jose!
```

Great! Note how we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator:

Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

In [20]:
```
def new_decorator(func):

    def wrap_func():
        print("Code would be here, before executing the func")

        func()

        print("Code here will execute after the func()")
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 8_Module.ipynb

naravishal Python Notes

1 contributor

648 lines (648 sloc) | 15.4 KB

Decorators

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions. Make sure to run every cell in this Notebook for this lecture to look the same on your own computer.

So let's break down the steps:

Functions Review

```
In [1]: def func():
            return 1
```

```
In [2]: func()
```

```
Out[2]: 1
```

Scope Review

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

```
In [ ]: s = 'Global Variable'

        def check_for_locals():
            print(locals())
```

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the `locals()` and `globals()` functions. For example:

```
In [ ]: print(globals())
```

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

```
In [ ]: print(globals().keys())
```

Note how **s** is there, the Global Variable we defined as a string:

```
In [ ]: globals()['s']
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

```
In [ ]: check_for_locals()
```

Great! Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions. Lets start with some simple examples:

```
In [3]: def hello(name='Jose'):
            return 'Hello '+name
```

```
In [4]: hello()
```

```
Out[4]: 'Hello Jose'
```

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

```
In [5]: greet = hello
```

```
In [6]: greet
```

```
Out[6]: <function __main__.hello>
```

```
In [7]: greet()
```

```
Out[7]: 'Hello Jose'
```

So what happens when we delete the name **hello**?

```
In [8]: del hello
```

```
In [9]: hello()
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-9-a75d7781aaeb> in <module>()
----> 1 hello()

NameError: name 'hello' is not defined
```

```
In [10]: greet()
```

```
Out[10]: 'Hello Jose'
```

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

Functions within functions

Great! So we've seen how we can treat functions as objects, now let's see how we can define functions inside of other functions:

```
In [11]: def hello(name='Jose'):
    print('The hello() function has been executed')
    
    def greet():
        return '\t This is inside the greet() function'
    
    def welcome():
        return "\t This is inside the welcome() function"
    
    print(greet())
    print(welcome())
    print("Now we are back inside the hello() function")
```

In [12]:
```
hello()
```

```
The hello() function has been executed
	 This is inside the greet() function
	 This is inside the welcome() function
Now we are back inside the hello() function
```

In [13]:
```
welcome()
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-13-a401d7101853> in <module>()
----> 1 welcome()

NameError: name 'welcome' is not defined
```

Note how due to scope, the welcome() function is not defined outside of the hello() function. Now lets learn about returning functions from within functions:

Returning Functions

In [14]:
```
def hello(name='Jose'):
    
    def greet():
        return '\t This is inside the greet() function'
    
    def welcome():
        return "\t This is inside the welcome() function"
    
    if name == 'Jose':
        return greet
    else:
        return welcome
```

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

In [15]:
```
x = hello()
```

In [16]:
```
x
```

Out[16]:
```
<function __main__.hello.<locals>.greet>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

In [17]:
```
print(x())
```

```
                This is inside the greet() function
```

Let's take a quick look at the code again.

In the `if/else` clause we are returning `greet` and `welcome`, not `greet()` and `welcome()`.

This is because when you put a pair of parentheses after it, the function gets executed; whereas if you don’t put parentheses after it, then it can be passed around and can be assigned to other variables without executing it.

When we write `x = hello()`, hello() gets executed and because the name is Jose by default, the function `greet` is returned. If we change the statement to `x = hello(name = "Sam")` then the `welcome` function will be returned. We can also do `print(hello()())` which outputs *This is inside the greet() function*.

Functions as Arguments

Now let's see how we can pass functions as arguments into other functions:

In [18]:
```
def hello():
    return 'Hi Jose!'

def other(func):
    print('Other code would go here')
    print(func())
```

In [19]:
```
other(hello)
```

```
Other code would go here
Hi Jose!
```

Great! Note how we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator:

Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

In [20]:
```
def new_decorator(func):

    def wrap_func():
        print("Code would be here, before executing the func")

        func()

        print("Code here will execute after the func()")
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 9_Module_2.ipynb

naravishal Python Notes

1 contributor

511 lines (511 sloc) | 11 KB

Collections Module

The collections module is a built-in module that implements specialized container data types providing alternatives to Python’s general purpose built-in containers. We've already gone over the basics: dict, list, set, and tuple.

Now we'll learn about the alternatives that the collections module provides.

Counter

Counter is a *dict* subclass which helps count hashable objects. Inside of it elements are stored as dictionary keys and the counts of the objects are stored as the value.

Let's see how it can be used:

```
In [1]: from collections import Counter
```

Counter() with lists

```
In [2]: lst = [1,2,2,2,2,3,3,3,1,2,1,12,3,2,32,1,21,1,223,1]

        Counter(lst)

Out[2]: Counter({1: 6, 2: 6, 3: 4, 12: 1, 21: 1, 32: 1, 223: 1})
```

Counter with strings

```
In [3]: Counter('aabsbsbsbhshhbbsbs')

Out[3]: Counter({'a': 2, 'b': 7, 'h': 3, 's': 6})
```

Counter with words in a sentence

```
In [4]: s = 'How many times does each word show up in this sentence word tim
        es each each word'

        words = s.split()

        Counter(words)

Out[4]: Counter({'How': 1,
                 'does': 1,
                 'each': 3,
                 'in': 1,
                 'many': 1,
                 'sentence': 1,
                 'show': 1,
                 'this': 1,
                 'times': 2,
                 'up': 1,
                 'word': 3})
```

```
In [5]: # Methods with Counter()
        c = Counter(words)
```

```
c = Counter(words)

c.most_common(2)
```

Out[5]:
```
[('each', 3), ('word', 3)]
```

Common patterns when using the Counter() object

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pair
s
c.most_common()[:-n-1:-1]       # n least common elements
c += Counter()                  # remove zero and negative counts
```

defaultdict

defaultdict is a dictionary-like object which provides all methods provided by a dictionary but takes a first argument (default_factory) as a default data type for the dictionary. Using defaultdict is faster than doing the same using dict.set_default method.

A defaultdict will never raise a KeyError. Any key that does not exist gets the value returned by the default factory.

In [6]:
```
from collections import defaultdict
```

In [7]:
```
d = {}
```

In [8]:
```
d['one']
```

```
---------------------------------------------------------------------
-------
KeyError                                  Traceback (most recent cal
l last)
<ipython-input-8-07706fc5dc20> in <module>()
----> 1 d['one']

KeyError: 'one'
```

In [9]:
```
d  = defaultdict(object)
```

In [10]:
```
d['one']
```

Out[10]:
```
<object at 0x216de27bcf0>
```

In [11]:
```
for item in d:
    print(item)
```

```
one
```

Can also initialize with default values:

```
In [12]: d = defaultdict(lambda: 0)
```

```
In [13]: d['one']
```

```
Out[13]: 0
```

namedtuple

The standard tuple uses numerical indexes to access its members, for example:

```
In [14]: t = (12,13,14)
```

```
In [15]: t[0]
```

```
Out[15]: 12
```

For simple use cases, this is usually enough. On the other hand, remembering which index should be used for each value can lead to errors, especially if the tuple has a lot of fields and is constructed far from where it is used. A namedtuple assigns names, as well as the numerical index, to each member.

Each kind of namedtuple is represented by its own class, created by using the namedtuple() factory function. The arguments are the name of the new class and a string containing the names of the elements.

You can basically think of namedtuples as a very quick way of creating a new object/class type with some attribute fields. For example:

```
In [22]: from collections import namedtuple
```

```
In [23]: Dog = namedtuple('Dog',['age','breed','name'])

         sam = Dog(age=2,breed='Lab',name='Sammy')

         frank = Dog(age=2,breed='Shepard',name="Frankie")
```

We construct the namedtuple by first passing the object type name (Dog) and then passing a string with the variety of fields as a string with spaces between the field names. We can then call on the various attributes:

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 9_Module_3.ipynb

naravishal Python Notes

1 contributor

280 lines (280 sloc) | 6.53 KB

datetime module

Python has the datetime module to help deal with timestamps in your code. Time values are represented with the time class. Times have attributes for hour, minute, second, and microsecond. They can also include time zone information. The arguments to initialize a time instance are optional, but the default of 0 is unlikely to be what you want.

time

Let's take a look at how we can extract time information from the datetime module. We can create a timestamp by specifying datetime.time(hour,minute,second,microsecond)

In [1]:
```
import datetime

t = datetime.time(4, 20, 1)

# Let's show the different components
print(t)
print('hour  :', t.hour)
print('minute:', t.minute)
print('second:', t.second)
print('microsecond:', t.microsecond)
print('tzinfo:', t.tzinfo)
```

```
04:20:01
hour  : 4
minute: 20
second: 1
microsecond: 0
tzinfo: None
```

Note: A time instance only holds values of time, and not a date associated with the time.

We can also check the min and max values a time of day can have in the module:

In [2]:
```
print('Earliest  :', datetime.time.min)
print('Latest    :', datetime.time.max)
print('Resolution:', datetime.time.resolution)
```

```
Earliest  : 00:00:00
Latest    : 23:59:59.999999
Resolution: 0:00:00.000001
```

The min and max class attributes reflect the valid range of times in a single day.

Dates

datetime (as you might suspect) also allows us to work with date timestamps. Calendar date values are represented with the date class. Instances have attributes for year, month, and day. It is easy to create a date representing today’s date using the today() class method.

Let's see some examples:

```
In [3]: today = datetime.date.today()
        print(today)
        print('ctime:', today.ctime())
        print('tuple:', today.timetuple())
        print('ordinal:', today.toordinal())
        print('Year :', today.year)
        print('Month:', today.month)
        print('Day  :', today.day)
```

```
2020-06-10
ctime: Wed Jun 10 00:00:00 2020
tuple: time.struct_time(tm_year=2020, tm_mon=6, tm_mday=10, tm_hour=
0, tm_min=0, tm_sec=0, tm_wday=2, tm_yday=162, tm_isdst=-1)
ordinal: 737586
Year : 2020
Month: 6
Day  : 10
```

As with time, the range of date values supported can be determined using the min and max attributes.

```
In [4]: print('Earliest  :', datetime.date.min)
        print('Latest    :', datetime.date.max)
        print('Resolution:', datetime.date.resolution)
```

```
Earliest  : 0001-01-01
Latest    : 9999-12-31
Resolution: 1 day, 0:00:00
```

Another way to create new date instances uses the replace() method of an existing date. For example, you can change the year, leaving the day and month alone.

```
In [5]: d1 = datetime.date(2015, 3, 11)
        print('d1:', d1)

        d2 = d1.replace(year=1990)
        print('d2:', d2)
```

```
d1: 2015-03-11
d2: 1990-03-11
```

Arithmetic

We can perform arithmetic on date objects to check for time differences. For example:

```
In [6]: d1

Out[6]: datetime.date(2015, 3, 11)

In [7]: d2
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 9_Module_4.ipynb

naravishal Python Notes

1 contributor

1197 lines (1197 sloc) | 26.5 KB

Math and Random Modules

Python comes with a built in math module and random module. In this lecture we will give a brief tour of their capabilities. Usually you can simply look up the function call you are looking for in the online documentation.

- Math Module (https://docs.python.org/3/library/math.html)
- Random Module (https://docs.python.org/3/library/random.html)

We won't go through every function available in these modules since there are so many, but we will show some useful ones.

Useful Math Functions

In [3]:
```python
import math
```

In [9]:
```python
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the inverse hyperbolic sine of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
        atan2(y, x)
        Return the arc tangent (measured in radians) of y/x.
        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(...)
        atanh(x)

        Return the inverse hyperbolic tangent of x.

    ceil(...)
        ceil(x)

        Return the ceiling of x as an Integral.
        This is the smallest integer >= x.

    copysign(...)
        copysign(x, y)

        Return a float with the magnitude (absolute value) of x but
the sign
        of y. On platforms that support signed zeros, copysign(1.0,
-0.0)
        returns -1.0.

    cos(...)
        cos(x)

        Return the cosine of x (measured in radians).

    cosh(...)
        cosh(x)

        Return the hyperbolic cosine of x.

    degrees(...)
        degrees(x)

        Convert angle x from radians to degrees.

    erf(...)
        erf(x)

        Error function at x.

    erfc(...)
        erfc(x)

        Complementary error function at x.

    exp(...)
        exp(x)

        Return e raised to the power of x.

    expm1(...)
        expm1(x)

        Return exp(x)-1.
        This function avoids the loss of precision involved in the d
irect evaluation of exp(x)-1 for small x.

    fabs(...)
```

```
        fabs(x)

        Return the absolute value of the float x.

    factorial(...)
        factorial(x) -> Integral

        Find x!. Raise a ValueError if x is negative or non-integra
l.

    floor(...)
        floor(x)

        Return the floor of x as an Integral.
        This is the largest integer <= x.

    fmod(...)
        fmod(x, y)

        Return fmod(x, y), according to platform C.  x % y may diffe
r.

    frexp(...)
        frexp(x)

        Return the mantissa and exponent of x, as pair (m, e).
        m is a float and e is an int, such that x = m * 2.**e.
        If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.

    fsum(...)
        fsum(iterable)

        Return an accurate floating point sum of values in the itera
ble.
        Assumes IEEE-754 floating point arithmetic.

    gamma(...)
        gamma(x)

        Gamma function at x.

    gcd(...)
        gcd(x, y) -> int
        greatest common divisor of x and y

    hypot(...)
        hypot(x, y)

        Return the Euclidean distance, sqrt(x*x + y*y).

    isclose(...)
        isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0) -> bool

        Determine whether two floating point numbers are close in va
lue.

           rel_tol
               maximum difference for being considered "close", rela
tive to the
               magnitude of the input values
            abs_tol
               maximum difference for being considered "close", nega
```

```
                  maximum difference for being considered "close", rega
rdless of the
                  magnitude of the input values

        Return True if a is close in value to b, and False otherwis
e.

        For the values to be considered close, the difference betwee
n them
        must be smaller than at least one of the tolerances.

        -inf, inf and NaN behave similarly to the IEEE 754 Standard.
That
        is, NaN is not close to anything, even itself.  inf and -inf
are
        only close to themselves.

    isfinite(...)
        isfinite(x) -> bool

        Return True if x is neither an infinity nor a NaN, and False
otherwise.

    isinf(...)
        isinf(x) -> bool

        Return True if x is a positive or negative infinity, and Fal
se otherwise.

    isnan(...)
        isnan(x) -> bool

        Return True if x is a NaN (not a number), and False otherwis
e.

    ldexp(...)
        ldexp(x, i)

        Return x * (2**i).

    lgamma(...)
        lgamma(x)

        Natural logarithm of absolute value of Gamma function at x.

    log(...)
        log(x[, base])

        Return the logarithm of x to the given base.
        If the base not specified, returns the natural logarithm (ba
se e) of x.

    log10(...)
        log10(x)

        Return the base 10 logarithm of x.

    log1p(...)
        log1p(x)

        Return the natural logarithm of 1+x (base e).
        The result is computed in a way which is accurate for x near
```

```
zero.

    log2(...)
        log2(x)

        Return the base 2 logarithm of x.

    modf(...)
        modf(x)

        Return the fractional and integer parts of x.  Both results 
carry the sign
        of x and are floats.

    pow(...)
        pow(x, y)

        Return x**y (x to the power of y).

    radians(...)
        radians(x)

        Convert angle x from degrees to radians.

    sin(...)
        sin(x)

        Return the sine of x (measured in radians).

    sinh(...)
        sinh(x)

        Return the hyperbolic sine of x.

    sqrt(...)
        sqrt(x)

        Return the square root of x.

    tan(...)
        tan(x)

        Return the tangent of x (measured in radians).

    tanh(...)
        tanh(x)

        Return the hyperbolic tangent of x.

    trunc(...)
        trunc(x:Real) -> Integral

        Truncates x to the nearest Integral toward 0. Uses the __tru
nc__ magic method.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586
```

```
FILE
    (built-in)
```

Rounding Numbers

```
In [5]: value = 4.35
```

```
In [6]: math.floor(value)
```

```
Out[6]: 4
```

```
In [7]: math.ceil(value)
```

```
Out[7]: 5
```

```
In [8]: round(value)
```

```
Out[8]: 4
```

Mathematical Constants

```
In [20]: math.pi
```

```
Out[20]: 3.141592653589793
```

```
In [21]: from math import pi
```

```
In [22]: pi
```

```
Out[22]: 3.141592653589793
```

```
In [23]: math.e
```

```
Out[23]: 2.718281828459045
```

```
In [24]: math.tau
```

```
Out[24]: 6.283185307179586
```

```
In [25]: math.inf
```

```
Out[25]: inf
```

```
In [26]: math.nan
```

```
Out[26]: nan
```

Logarithmic Values

```
In [10]: math.e
```

```
Out[10]: 2.718281828459045
```

```
In [15]: # Log Base e
         math.log(math.e)

Out[15]: 1.0

In [12]: # Will produce an error if value does not exist mathmatically
         math.log(0)

         ---------------------------------------------------------------------
         -------
         ValueError                                Traceback (most recent cal
         l last)
         <ipython-input-12-7563e0a48092> in <module>()
         ----> 1 math.log(0)

         ValueError: math domain error

In [13]: math.log(10)

Out[13]: 2.302585092994046

In [17]: math.e ** 2.302585092994046

Out[17]: 10.000000000000002
```

Custom Base

```
In [18]: # math.log(x,base)
         math.log(100,10)

Out[18]: 2.0

In [19]: 10**2

Out[19]: 100
```

Trigonometrics Functions

```
In [30]: # Radians
         math.sin(10)

Out[30]: -0.5440211108893698

In [31]: math.degrees(pi/2)

Out[31]: 90.0

In [32]: math.radians(180)

Out[32]: 3.141592653589793
```

Random Module

Random Module allows us to create random numbers. We can even set a seed to produce the same random set every time

The explanation of how a computer attempts to generate random numbers is beyond the scope of this course since it involves higher level mathmatics. But if you are interested in this topic check out:

- https://en.wikipedia.org/wiki/Pseudorandom_number_generator (https://en.wikipedia.org/wiki/Pseudorandom_number_generator)
- https://en.wikipedia.org/wiki/Random_seed (https://en.wikipedia.org/wiki/Random_seed)

Understanding a seed

Setting a seed allows us to start from a seeded psuedorandom number generator, which means the same random numbers will show up in a series. Note, you need the seed to be in the same cell if your using jupyter to guarantee the same results each time. Getting a same set of random numbers can be important in situations where you will be trying different variations of functions and want to compare their performance on random values, but want to do it fairly (so you need the same set of random numbers each time).

```
In [34]: import random
```

```
In [41]: random.randint(0,100)

Out[41]: 62
```

```
In [42]: random.randint(0,100)

Out[42]: 10
```

```
In [45]: # The value 101 is completely arbitrary, you can pass in any number
          you want
         random.seed(101)
         # You can run this cell as many times as you want, it will always re
         turn the same number
         random.randint(0,100)

Out[45]: 74
```

```
In [46]: random.randint(0,100)

Out[46]: 24
```

```
In [48]: # The value 101 is completely arbitrary, you can pass in any number
          you want
         random.seed(101)
         print(random.randint(0,100))
         print(random.randint(0,100))
         print(random.randint(0,100))
         print(random.randint(0,100))
         print(random.randint(0,100))

         74
         24
         69
         45
         59
```

Random Integers

```
In [49]: random.randint(0,100)
```

```
Out[49]: 6
```

Random with Sequences

Grab a random item from a list

```
In [70]: mylist = list(range(0,20))
```

```
In [71]: mylist
```

```
Out[71]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
         9]
```

```
In [72]: random.choice(mylist)
```

```
Out[72]: 12
```

```
In [73]: mylist
```

```
Out[73]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
         9]
```

Sample with Replacement

Take a sample size, allowing picking elements more than once. Imagine a bag of numbered lottery balls, you reach in to grab a random lotto ball, then after marking down the number, **you place it back in the bag**, then continue picking another one.

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 9_Module_5.ipynb

naravishal Python Notes

1 contributor

1274 lines (1274 sloc) 26.7 KB

Overview of Regular Expressions

Regular Expressions (sometimes called regex for short) allows a user to search for strings using almost any sort of rule they can come up. For example, finding all capital letters in a string, or finding a phone number in a document.

Regular expressions are notorious for their seemingly strange syntax. This strange syntax is a byproduct of their flexibility. Regular expressions have to be able to filter out any string pattern you can imagine, which is why they have a complex string pattern format.

Let's begin by explaining how to search for basic patterns in a string!

Searching for Basic Patterns

Let's imagine that we have the following string:

```
In [1]: text = "The person's phone number is 408-555-1234. Call soon!"
```

We'll start off by trying to find out if the string "phone" is inside the text string. Now we could quickly do this with:

```
In [2]: 'phone' in text

Out[2]: True
```

But let's show the format for regular expressions, because later on we will be searching for patterns that won't have such a simple solution.

```
In [3]: import re
```

```
In [4]: pattern = 'phone'
```

```
In [5]: re.search(pattern,text)

Out[5]: <_sre.SRE_Match object; span=(13, 18), match='phone'>
```

```
In [6]: pattern = "NOT IN TEXT"
```

```
In [7]: re.search(pattern,text)
```

Now we've seen that re.search() will take the pattern, scan the text, and then returns a Match object. If no pattern is found, a None is returned (in Jupyter Notebook this just means that nothing is output below the cell).

Let's take a closer look at this Match object.

```
In [8]: pattern = 'phone'
In [9]: match = re.search(pattern,text)
```

```
In [10]: match
Out[10]: <_sre.SRE_Match object; span=(13, 18), match='phone'>
```

Notice the span, there is also a start and end index information.

```
In [11]: match.span()
Out[11]: (13, 18)
```

```
In [12]: match.start()
Out[12]: 13
```

```
In [13]: match.end()
Out[13]: 18
```

But what if the pattern occurs more than once?

```
In [14]: text = "my phone is a new phone"
```

```
In [15]: match = re.search("phone",text)
```

```
In [16]: match.span()
Out[16]: (3, 8)
```

Notice it only matches the first instance. If we wanted a list of all matches, we can use .findall() method:

```
In [17]: matches = re.findall("phone",text)
```

```
In [18]: matches
Out[18]: ['phone', 'phone']
```

```
In [19]: len(matches)
Out[19]: 2
```

To get actual match objects, use the iterator:

```
In [20]: for match in re.finditer("phone",text):
             print(match.span())

         (3, 8)
         (18, 23)
```

If you wanted the actual text that matched, you can use the .group() method.

```
In [21]: match.group()
```

```
Out[21]: 'phone'
```

Patterns

So far we've learned how to search for a basic string. What about more complex examples? Such as trying to find a telephone number in a large string of text? Or an email address?

We could just use search method if we know the exact phone or email, but what if we don't know it? We may know the general format, and we can use that along with regular expressions to search the document for strings that match a particular pattern.

This is where the syntax may appear strange at first, but take your time with this, often its just a matter of looking up the pattern code.

Let' begin!

Identifiers for Characters in Patterns

Characters such as a digit or a single string have different codes that represent them. You can use these to build up a pattern string. Notice how these make heavy use of the backwards slash \ . Because of this when defining a pattern string for regular expression we use the format:

```
r'mypattern'
```

placing the r in front of the string allows python to understand that the \ in the pattern string are not meant to be escape slashes.

Below you can find a table of all the possible identifiers:

</table>

For example:

```
In [22]: text = "My telephone number is 408-555-1234"
```

```
In [23]: phone = re.search(r'\d\d\d-\d\d\d-\d\d\d\d',text)
```

```
In [24]: phone.group()
Out[24]: '408-555-1234'
```

Notice the repetition of \d. That is a bit of an annoyance, especially if we are looking for very long strings of numbers. Let's explore the possible quantifiers.

Quantifiers

Now that we know the special character designations, we can use them along with quantifiers to define how many we expect.

Character	Description	Example Pattern Code	Exammple Match
\d	A digit	file_\d\d	file_25
\w	Alphanumeric	\w-\w\w\w	A-b_1
\s	White space	a\sb\sc	a b c
\D	A non digit	\D\D\D	ABC
\W	Non-alphanumeric	\W\W\W\W\W	*-+=)
\S	Non-whitespace	\S\S\S\S	Yoyo

</table>

Let's rewrite our pattern using these quantifiers:

```
In [25]: re.search(r'\d{3}-\d{3}-\d{4}',text)

Out[25]: <_sre.SRE_Match object; span=(23, 35), match='408-555-1234'>
```

Groups

What if we wanted to do two tasks, find phone numbers, but also be able to quickly extract their area code (the first three digits). We can use groups for any general task that involves grouping together regular expressions (so that we can later break them down).

Using the phone number example, we can separate groups of regular expressions using parenthesis:

```
In [26]: phone_pattern = re.compile(r'(\d{3})-(\d{3})-(\d{4})')

In [27]: results = re.search(phone_pattern,text)

In [28]: # The entire result
         results.group()

Out[28]: '408-555-1234'

In [29]: # Can then also call by group position.
         # remember groups were separated by parenthesis ()
         # Something to note is that group ordering starts at 1. Passing in
         0 returns everything
         results.group(1)

Out[29]: '408'

In [30]: results.group(2)

Out[30]: '555'

In [31]: results.group(3)

Out[31]: '1234'

In [32]: # We only had three groups of parenthesis
```

```
results.group(4)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-32-866de7a94a57> in <module>()
      1 # We only had three groups of parenthesis
----> 2 results.group(4)

IndexError: no such group
```

Additional Regex Syntax

Or operator |

Use the pipe operator to have an **or** statment. For example

```
In [33]: re.search(r"man|woman","This man was here.")
```

```
Out[33]: <_sre.SRE_Match object; span=(5, 8), match='man'>
```

```
In [34]: re.search(r"man|woman","This woman was here.")
```

```
Out[34]: <_sre.SRE_Match object; span=(5, 10), match='woman'>
```

The Wildcard Character

Use a "wildcard" as a placement that will match any character placed there. You can use a simple period **.** for this. For example:

```
In [35]: re.findall(r".at","The cat in the hat sat here.")
```

```
Out[35]: ['cat', 'hat', 'sat']
```

```
In [36]: re.findall(r".at","The bat went splat")
```

```
Out[36]: ['bat', 'lat']
```

Notice how we only matched the first 3 letters, that is because we need a **.** for each wildcard letter. Or use the quantifiers described above to set its own rules.

```
In [37]: re.findall(r"...at","The bat went splat")
```

```
Out[37]: ['e bat', 'splat']
```

However this still leads the problem to grabbing more beforehand. Really we only want words that end with "at".

```
In [38]: # One or more non-whitespace that ends with 'at'
         re.findall(r'\S+at',"The bat went splat")
```

```
Out[38]: ['bat', 'splat']
```

Starts with and Ends With

We can use the **^** to signal starts with, and the **$** to signal ends with:

```
In [39]: # Ends with a number
         re.findall(r'\d$','This ends with a number 2')

Out[39]: ['2']

In [40]: # Starts with a number
         re.findall(r'^\d','1 is the loneliest number.')

Out[40]: ['1']
```

Note that this is for the entire string, not individual words!

Exclusion

To exclude characters, we can use the **^** symbol in conjunction with a set of brackets **[]**. Anything inside the brackets is excluded. For example:

```
In [41]: phrase = "there are 3 numbers 34 inside 5 this sentence."

In [42]: re.findall(r'[^\d]',phrase)

Out[42]: ['t',
          'h',
          'e',
          'r',
          'e',
          ' ',
          'a',
          'r',
          'e',
          ' ',
          ' ',
          'n',
          'u',
          'm',
          'b',
          'e',
          'r',
          's',
          ' ',
          ' ',
          'i',
          'n',
          's',
          'i',
          'd',
          'e',
          ' ',
          ' ',
          't',
          'h',
          'i',
          's',
          ' ',
 's',
 'e',
 'n',
 't',
 'e',
 'n',
 'c',
 'e',
 '.']
```

To get the words back together, use a + sign

```
In [43]: re.findall(r'[^\d]+',phrase)

Out[43]: ['there are ', ' numbers ', ' inside ', ' this sentence.']
```

We can use this to remove punctuation from a sentence.

```
In [44]: test_phrase = 'This is a string! But it has punctuation. How can w
         e remove it?'

In [45]: re.findall('[^!.? ]+',test_phrase)

Out[45]: ['This',
          'is',
          'a',
          'string',
          'But',
          'it',
          'has',
          'punctuation',
          'How',
          'can',
          'we',
          'remove',
          'it']

In [46]: clean = ' '.join(re.findall('[^!.? ]+',test_phrase))

In [47]: clean

Out[47]: 'This is a string But it has punctuation How can we remove it'
```

Brackets for Grouping

As we showed above we can use brackets to group together options, for example if we wanted to find hyphenated words:

```
In [48]: text = 'Only find the hypen-words in this sentence. But you do not
         know how long-ish they are'
```

naravishal / personal Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

personal / python / 9_Module_6.ipynb

naravishal Python Notes

1 contributor

429 lines (429 sloc) | 8.59 KB

Timing your code

Sometimes it's important to know how long your code is taking to run, or at least know if a particular line of code is slowing down your entire project. Python has a built-in timing module to do this.

Example Function or Script

Here we have two functions that do the same thing, but in different ways. How can we tell which one is more efficient? Let's time it!

```
In [10]: def func_one(n):
             '''
             Given a number n, returns a list of string integers
             ['0','1','2',...'n]
             '''
             return [str(num) for num in range(n)]
```

```
In [11]: func_one(10)

Out[11]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
In [12]: def func_two(n):
             '''
             Given a number n, returns a list of string integers
             ['0','1','2',...'n]
             '''
             return list(map(str,range(n)))
```

```
In [13]: func_two(10)

Out[13]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Timing Start and Stop

We can try using the time module to simply calculate the elapsed time for the code. Keep in mind, due to the time module's precision, the code needs to take **at least** 0.1 seconds to complete.

```
In [57]: import time
```

```
In [58]: # STEP 1: Get start time
         start_time = time.time()
         # Step 2: Run your code you want to time
         result = func_one(1000000)
         # Step 3: Calculate total time elapsed
         end_time = time.time() - start_time
```

```
In [59]: end_time

Out[59]: 0.18550348281860352
```

```
In [60]: # STEP 1: Get start time
         start_time = time.time()
```

```
# Step 2: Run your code you want to time
result = func_two(1000000)
# Step 3: Calculate total time elapsed
end_time = time.time() - start_time
```

In [61]:
```
end_time
```

Out[61]:
```
0.1496279239654541
```

Timeit Module

What if we have two blocks of code that are quite fast, the difference from the time.time() method may not be enough to tell which is fater. In this case, we can use the timeit module.

The timeit module takes in two strings, a statement (stmt) and a setup. It then runs the setup code and runs the stmt code some n number of times and reports back average length of time it took.

In [18]:
```
import timeit
```

The setup (anything that needs to be defined beforehand, such as def functions.)

In [39]:
```
setup = '''
def func_one(n):
    return [str(num) for num in range(n)]
'''
```

In [40]:
```
stmt = 'func_one(100)'
```

In [41]:
```
timeit.timeit(stmt,setup,number=100000)
```

Out[41]:
```
1.3161248000000114
```

Now let try running func_two 10,000 times and compare the length of time it took.

In [42]:
```
setup2 = '''
def func_two(n):
    return list(map(str,range(n)))
'''
```

In [43]:
```
stmt2 = 'func_two(100)'
```

In [44]:
```
timeit.timeit(stmt2,setup2,number=100000)
```

Out[44]:
```
1.0892171000000417
```

It looks like func_two is more efficient. You can specify more number of runs if you want to clarify the different for fast performing functions.

In [45]:
```
timeit.timeit(stmt,setup,number=1000000)
```

Out[45]:
```
13.129837899999984
```

In [46]:
```
timeit.timeit(stmt2,setup2,number=1000000)
```

```
Out[46]: 10.894090699999992
```

Timing you code with Jupyter "magic" method

NOTE: This method is ONLY available in Jupyter and the magic command needs to be at the top of the cell with nothing above it (not even commented code)

```
In [63]: %%timeit
         func_one(100)
```