# Introduction to Programming Using Python

Microsoft Python Certification | **Exam 98-381**

## Lecture 1:

# First Course in Python

Lecturer: **James W. Jiang**, Ph.D. | Summer, 2018

Microsoft™

SAVVY PRO
PROFESSIONAL EDUCATION TRAINING

# First Course in Python:

# Why Python

# First Course in Python

There are a LOT of different programming languages out there

Python is one of the easier ones to learn

There are lots of free tools out there you can use to code or learn Python

There are a lot of different ways to use Python code

# Does anyone really use Python?

Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.

Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

# First Course in Python:
# **Installation**

# Spyder Python

https://pythonhosted.org/spyder/installation.html

https://pythonhosted.org/spyder/

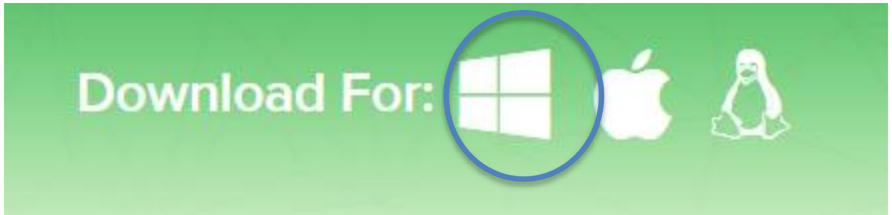
Install Spyder on Mac OSX:

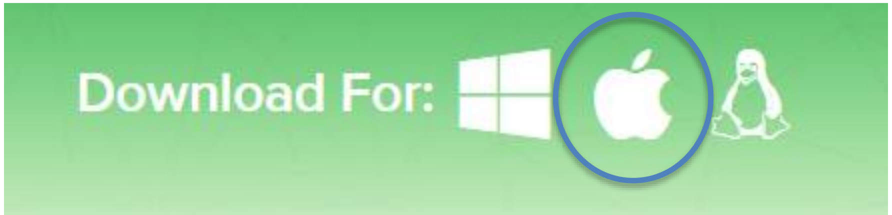
http://macappstore.org/spyder/

# Anaconda Python

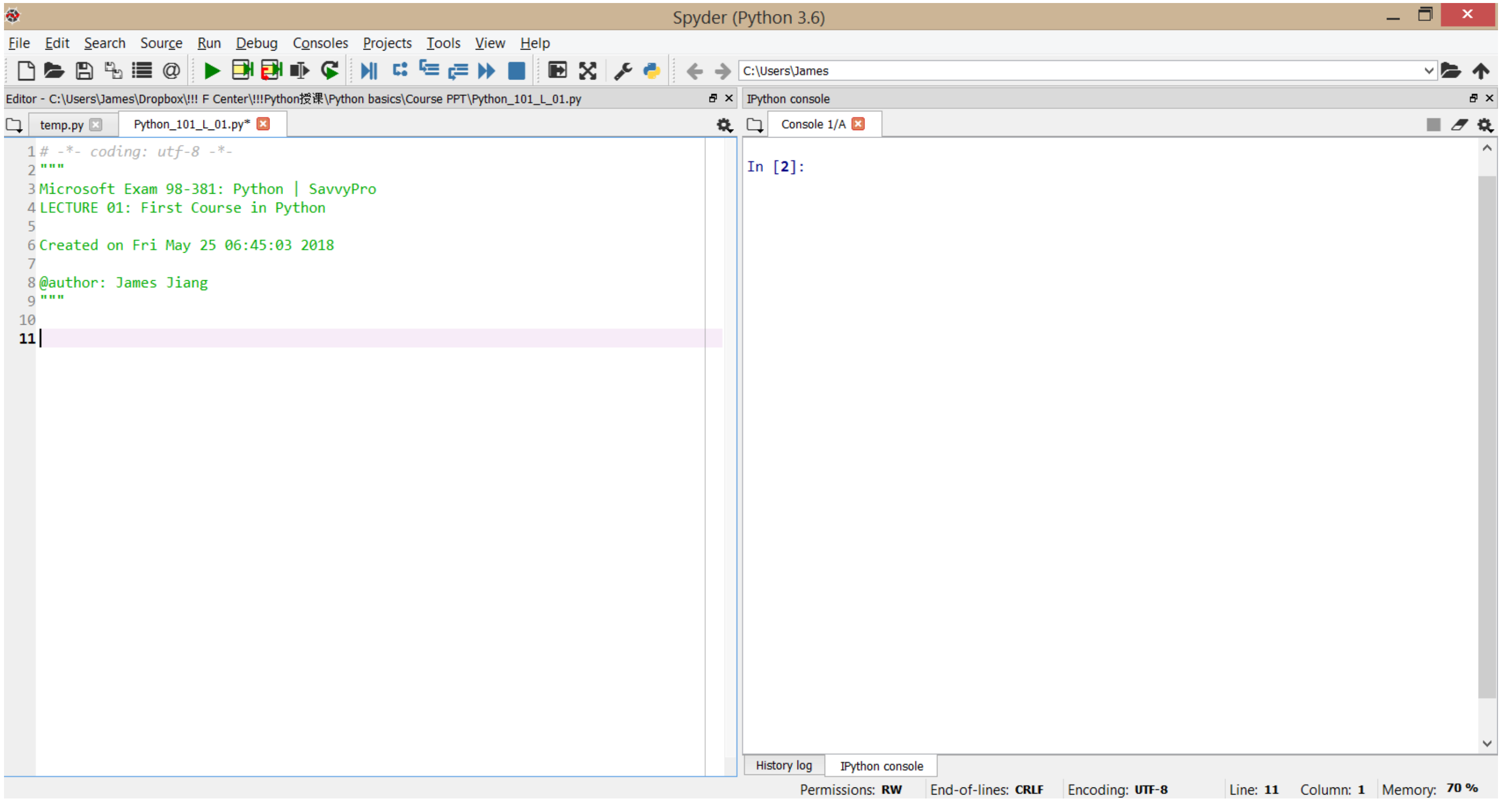
https://www.anaconda.com/download/

ANACONDA.





# User Interface



Editor - C:\Users\James\Dropbox\!!! F Center\!!!Python授课\Python basics\Course PPT\Python_101_L_01.py

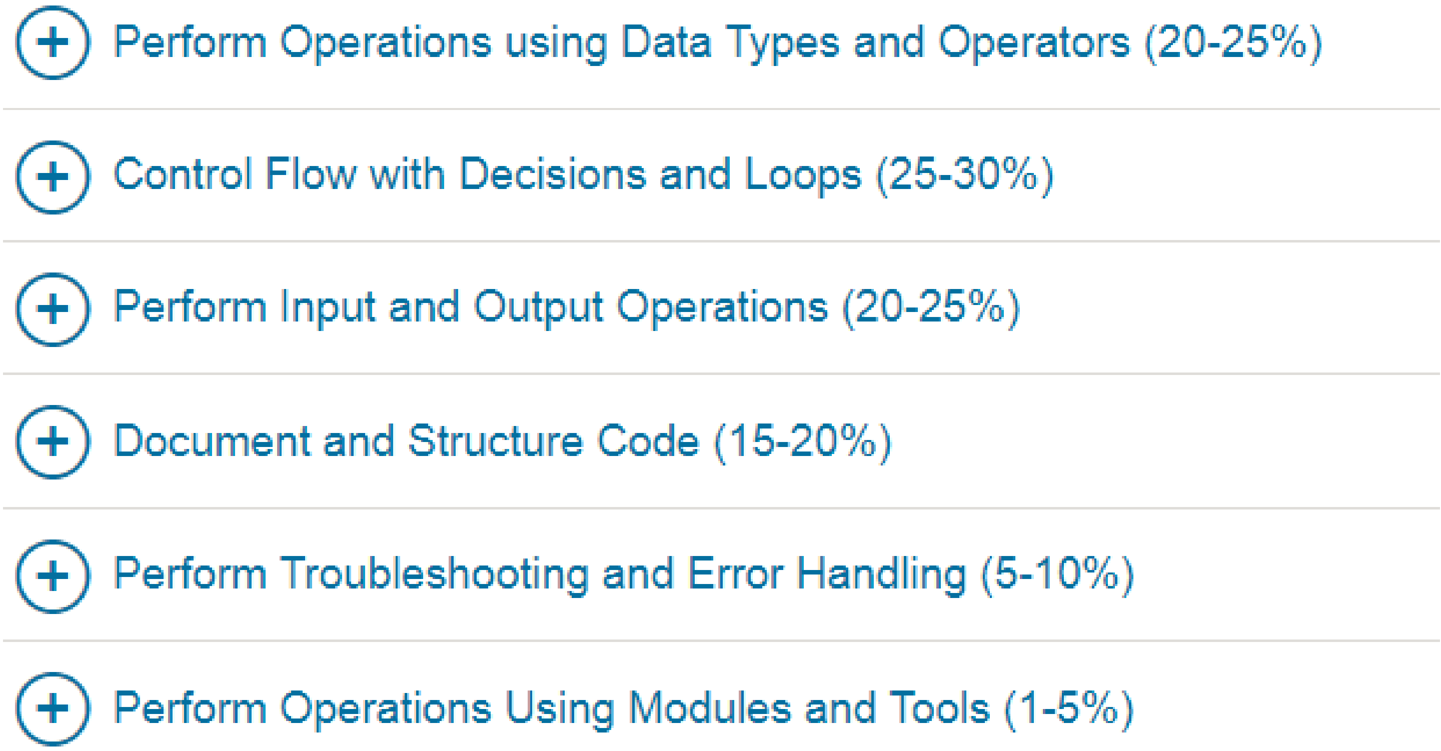
```
# -*- coding: utf-8 -*-
"""
Microsoft Exam 98-381: Python | SavvyPro
LECTURE 01: First Course in Python

Created on Fri May 25 06:45:03 2018

@author: James Jiang
"""

```

```
In [2]:
```

Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 11 Column: 1 Memory: 70 %

# First Course in Python:
# Microsoft Python Exam

# Outline



⊕ Perform Operations using Data Types and Operators (20-25%)

⊕ Control Flow with Decisions and Loops (25-30%)

⊕ Perform Input and Output Operations (20-25%)

⊕ Document and Structure Code (15-20%)

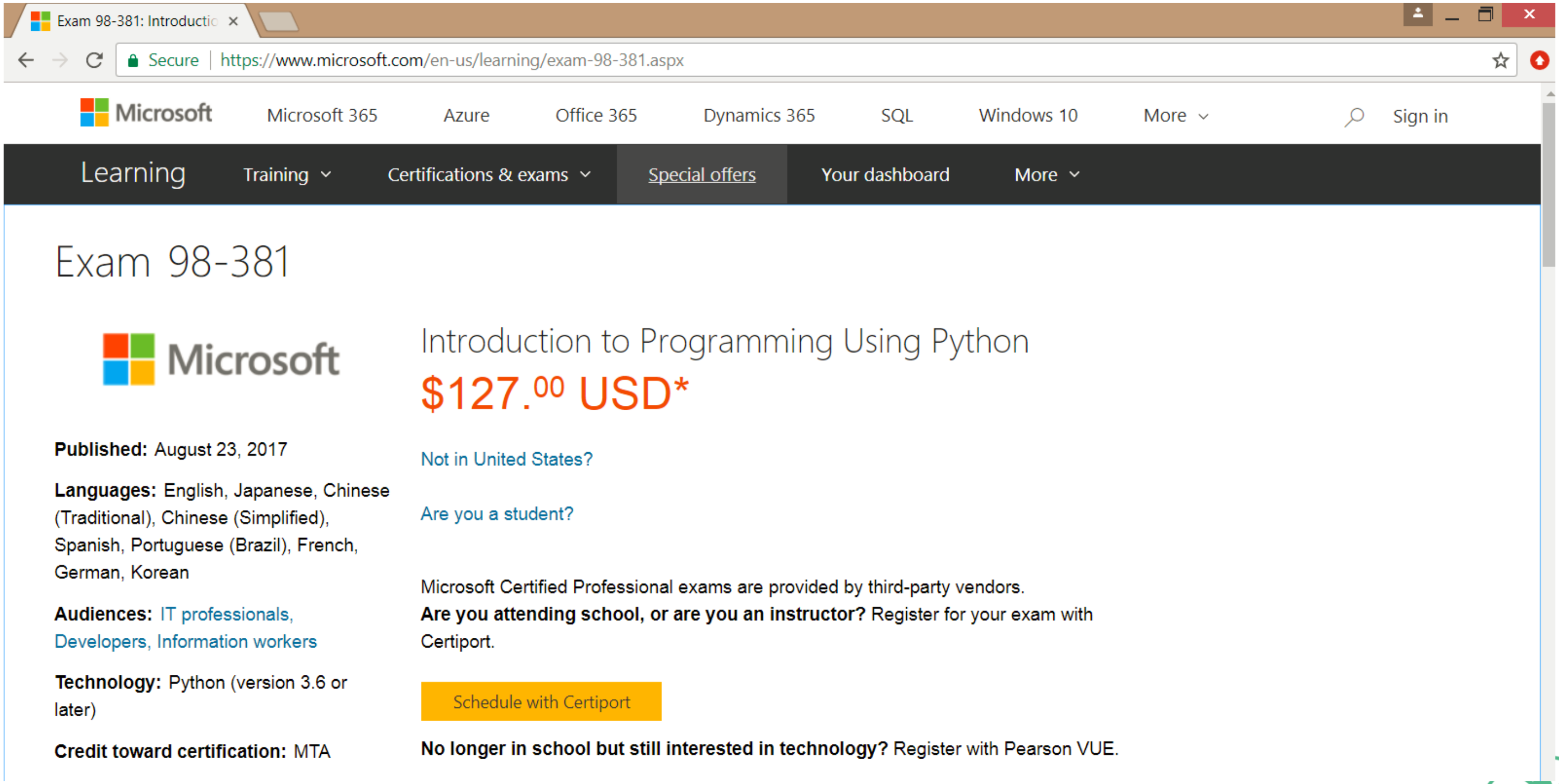
⊕ Perform Troubleshooting and Error Handling (5-10%)

⊕ Perform Operations Using Modules and Tools (1-5%)

Microsoft

# Microsoft Exam 98-381

https://www.microsoft.com/en-us/learning/exam-98-381.aspx



Exam 98-381

Introduction to Programming Using Python

$127.00 USD*

Not in United States?

Are you a student?

**Published:** August 23, 2017

**Languages:** English, Japanese, Chinese (Traditional), Chinese (Simplified), Spanish, Portuguese (Brazil), French, German, Korean

**Audiences:** IT professionals, Developers, Information workers

**Technology:** Python (version 3.6 or later)

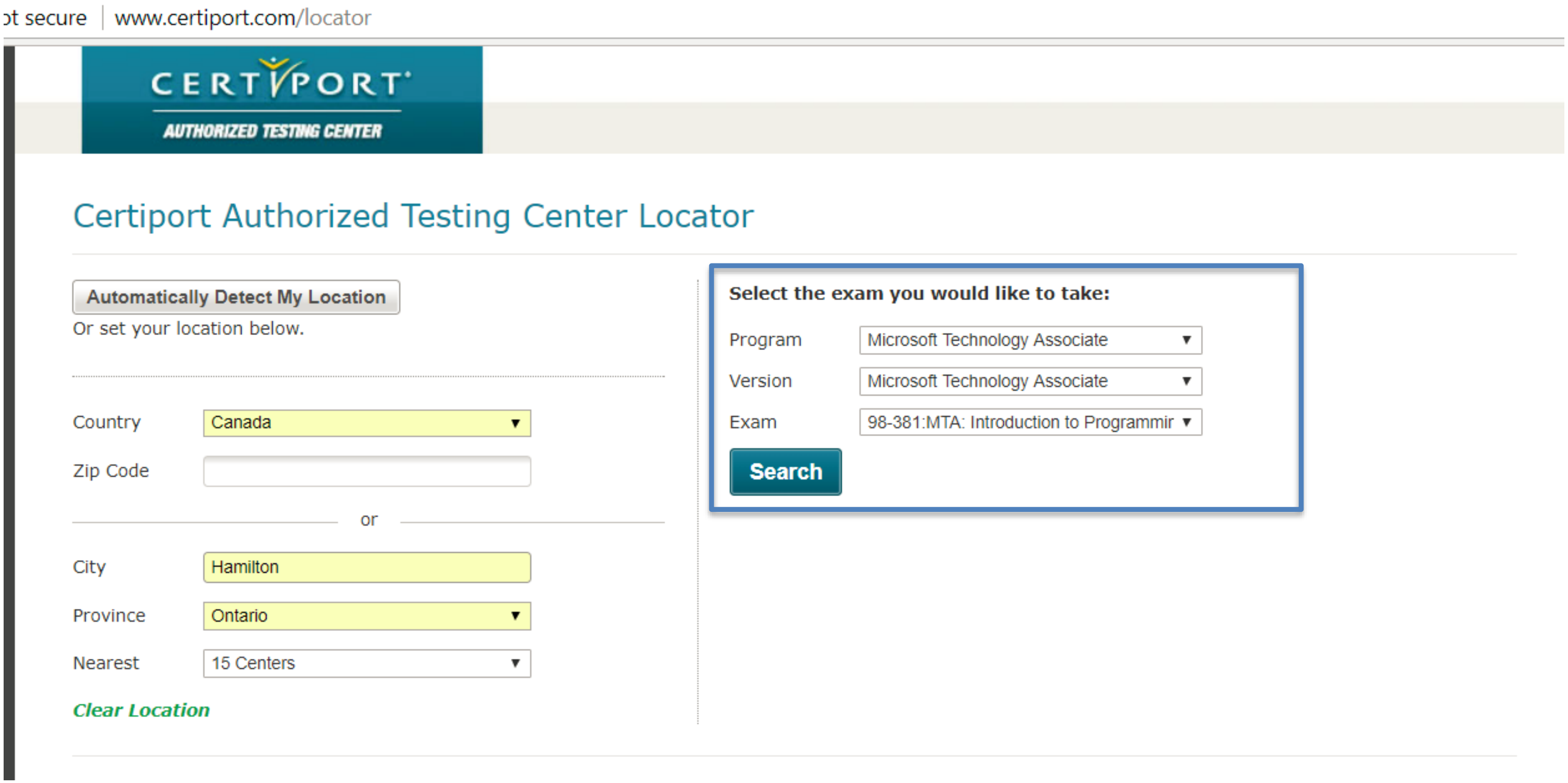
**Credit toward certification:** MTA

Microsoft Certified Professional exams are provided by third-party vendors.

**Are you attending school, or are you an instructor?** Register for your exam with Certiport.

**No longer in school but still interested in technology?** Register with Pearson VUE.

# Registration



ot secure | www.certiport.com/locator

CERTIPORT AUTHORIZED TESTING CENTER

## Certiport Authorized Testing Center Locator

Or set your location below.

| | |
|---|---|
| Country | Canada ▼ |
| Zip Code | |

or

| | |
|---|---|
| City | Hamilton |
| Province | Ontario ▼ |
| Nearest | 15 Centers ▼ |

**Select the exam you would like to take:**

| | |
|---|---|
| Program | Microsoft Technology Associate ▼ |
| Version | Microsoft Technology Associate ▼ |
| Exam | 98-381:MTA: Introduction to Programmir ▼ |

# First Course in Python:

# Displaying Text

# You can use single quotes or double quotes

```
print('Hickory Dickory Dock! The mouse ran up the clock')
print("Hickory Dickory Dock! The mouse ran up the clock")
```

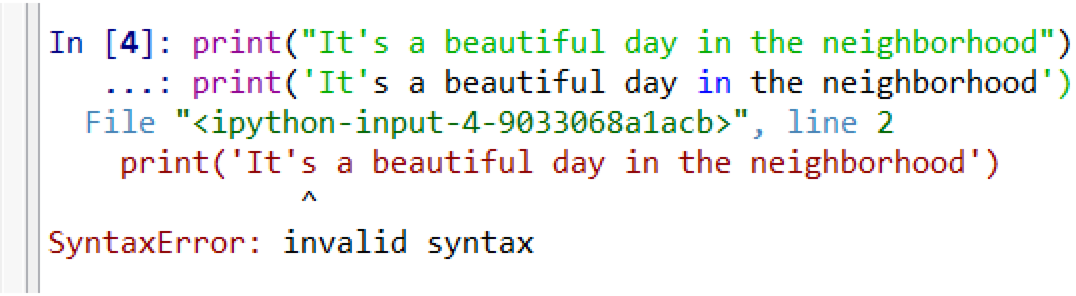
Trick: # run current line shortcut: F9 (Python Spyder)

# Does it matter if you use single or double quotes?

```
print("It's a beautiful day in the neighborhood")
print('It's a beautiful day in the neighborhood')
```

Only if the string you are displaying contains a single or double quote.

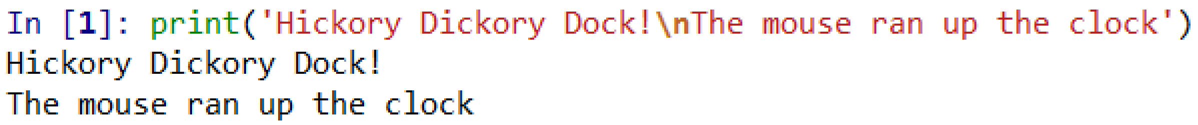
It’s a good habit to pick one and stick with it as much as possible.



```
In [4]: print("It's a beautiful day in the neighborhood")
   ...: print('It's a beautiful day in the neighborhood')
  File "<ipython-input-4-9033068a1acb>", line 2
    print('It's a beautiful day in the neighborhood')
             ^
SyntaxError: invalid syntax
```

# You can also use “\n” to force a new line

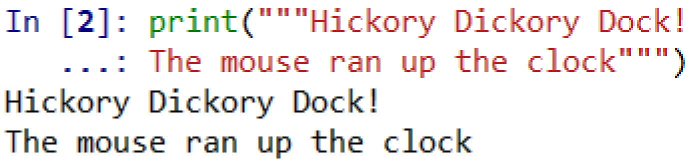
```
print('Hickory Dickory Dock!\nThe mouse ran up the
clock')
```



```
In [1]: print('Hickory Dickory Dock!\nThe mouse ran up the clock')
Hickory Dickory Dock!
The mouse ran up the clock
```

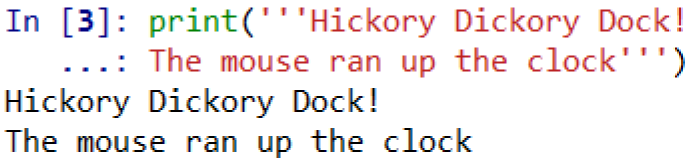
# Here’s a neat Python trick: triple quotes!

```
print("""Hickory Dickory Dock!
The mouse ran up the clock""")
```



```
In [2]: print("""Hickory Dickory Dock!
   ...: The mouse ran up the clock""")
Hickory Dickory Dock!
The mouse ran up the clock
```

```
print('''Hickory Dickory Dock!
The mouse ran up the clock''')
```



```
In [3]: print('''Hickory Dickory Dock!
   ...: The mouse ran up the clock''')
Hickory Dickory Dock!
The mouse ran up the clock
```

# Common Mistakes

```
print(Hickory Dickory Dock)
print('It's a small world')
print("Hi there')
prnit("Hello World!")

print('Hickory Dickory Dock')
print("It's a small world")
print("Hi there")
print("Hello World!")
```

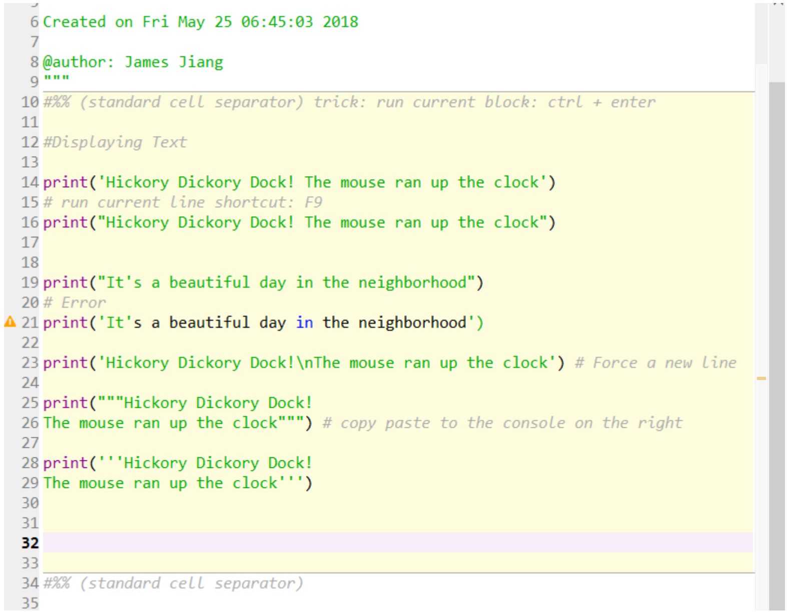
# First Course in Python:
# Intro to Formatting

# First Course in Python

use a # to indicate comments

```
#My first Python Application
#Created by me!
#Print command displays a message on the screen
print('Hello World’)
```

#%% (standard cell separator)



```
Created on Fri May 25 06:45:03 2018

@author: James Jiang
"""
#%% (standard cell separator) trick: run current block: ctrl + enter

#Displaying Text

print('Hickory Dickory Dock! The mouse ran up the clock')
# run current line shortcut: F9
print("Hickory Dickory Dock! The mouse ran up the clock")


print("It's a beautiful day in the neighborhood")
# Error
print('It's a beautiful day in the neighborhood')

print('Hickory Dickory Dock!\nThe mouse ran up the clock') # Force a new line

print("""Hickory Dickory Dock!
The mouse ran up the clock""") # copy paste to the console on the right

print('''Hickory Dickory Dock!
The mouse ran up the clock''')




#%% (standard cell separator)

```

# First Course in Python:
# Base Types

# Base Types

Base types:

- integer, float, Boolean, bytes
- string, list, tuple, set, dictionary

# Data Types

```
str_eg = 'this is a string'

fruits = ["apple", "mango", "orange"] #list

numbers = (1, 2, 3) #tuple

alphabets = {'a':'apple', 'b':'ball', 'c':'cat'}
#dictionary

vowels = {'a', 'e', 'i' , 'o', 'u'} #set

print(str_eg)

print(fruits)

print(numbers)

print(alphabets)

print(vowels)
```

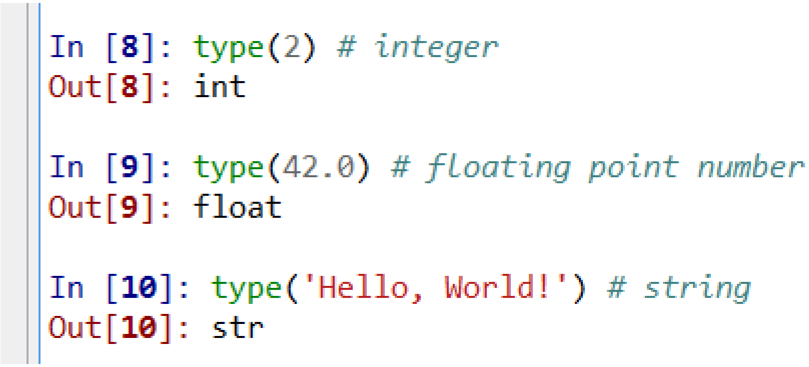
# Python Numbers

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as `int`, `float` and `complex` class in Python. We can use the `type()` function to know which class a variable or a value belongs to and the `isinstance()` function to check if an object belongs to a particular class.

```
a = 5

print(a, "is of type", type(a))

a = 2.0

print(a, "is of type", type(a))

a = 1+2j

print(a, "is complex number?", isinstance(1+2j,complex))
```

# Type Info

```
>>> type(2) # integer
<class 'int'>
>>> type(42.0) # floating point number
<class 'float'>
>>> type('Hello, World!’) # string
<class 'str'>
```



```
In [8]: type(2) # integer
Out[8]: int

In [9]: type(42.0) # floating point number
Out[9]: float

In [10]: type('Hello, World!') # string
Out[10]: str
```

# Practice Question

Please name the types of the following results from math operations.

```
type(2/1)
type(1+2)
type(2*1)
type(5/2)
type(2/1+1)
type(2/1*1)
```

# Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

```
>>> a = [1, 2.2, 'python']
```

We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts form 0 in Python.

# Python Tuple

Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.

It is defined within parentheses () where items are separated by commas.

```
>>> t = (5,'program', 1+3j)
```

We can use the slicing operator [] to extract items but we cannot change its value.

```
t = (5,'program', 1+3j)

print("t[1] = ", t[1])
```

# Python Strings

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

```
s = "This is a string"

s = '''a multiline

...

'''
```

Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

```
s = 'Hello world!'

s[5] # Strings are immutable in Python

s[5] ='d'
```

# Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4} # printing set variable

print("a = ", a)

print(type(a))
```

We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3}

a

{1, 2, 3}
```

# Python Dictionary

Dictionary is an unordered collection of key-value pairs. It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type. We use key to retrieve the respective value. But not the other way around.

```
>>> d = {1:'value','key':2}

>>> type(d)

<class 'dict'>
```

# First Course in Python:
# Integers

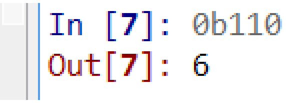
# Integer, int

```
101

0

-201

0b110 (binary)

0o112 (octal)

0x111 (hexa)
```

| Number System | Prefix |
|---|---|
| Binary | '0b' or '0B' |
| Octal | '0o' or '0O' |
| Hexadecimal | '0x' or '0X' |

# Integer, int (binary)

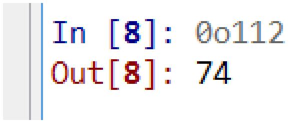
0b110



```
In [7]: 0b110
Out[7]: 6
```

# Integer, int (octal)

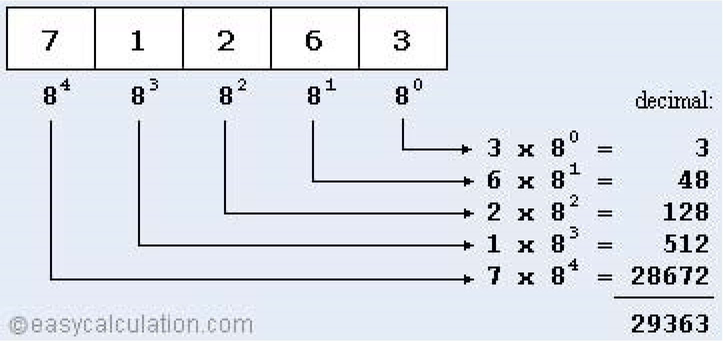
```
0o112 #octal
```



```
In [8]: 0o112
Out[8]: 74
```

$$112_8 = \mathbf{1} \times 8^2 + \mathbf{1} \times 8^1 + \mathbf{2} \times 8^0 = 74$$

```
0o71263 #octal
```

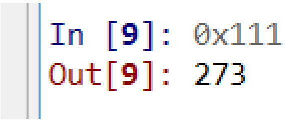


©easycalculation.com

# Integer, int (hexa)

## Hexadecimal

`0x111 (hexa)`



```
In [9]: 0x111
Out[9]: 273
```

# First Course in Python

| $0_{hex}$ | = | $0_{dec}$ | = | $0_{oct}$ | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| $1_{hex}$ | = | $1_{dec}$ | = | $1_{oct}$ | 0 | 0 | 0 | 1 |
| $2_{hex}$ | = | $2_{dec}$ | = | $2_{oct}$ | 0 | 0 | 1 | 0 |
| $3_{hex}$ | = | $3_{dec}$ | = | $3_{oct}$ | 0 | 0 | 1 | 1 |
| $4_{hex}$ | = | $4_{dec}$ | = | $4_{oct}$ | 0 | 1 | 0 | 0 |
| $5_{hex}$ | = | $5_{dec}$ | = | $5_{oct}$ | 0 | 1 | 0 | 1 |
| $6_{hex}$ | = | $6_{dec}$ | = | $6_{oct}$ | 0 | 1 | 1 | 0 |
| $7_{hex}$ | = | $7_{dec}$ | = | $7_{oct}$ | 0 | 1 | 1 | 1 |
| $8_{hex}$ | = | $8_{dec}$ | = | $10_{oct}$ | 1 | 0 | 0 | 0 |
| $9_{hex}$ | = | $9_{dec}$ | = | $11_{oct}$ | 1 | 0 | 0 | 1 |
| $A_{hex}$ | = | $10_{dec}$ | = | $12_{oct}$ | 1 | 0 | 1 | 0 |
| $B_{hex}$ | = | $11_{dec}$ | = | $13_{oct}$ | 1 | 0 | 1 | 1 |
| $C_{hex}$ | = | $12_{dec}$ | = | $14_{oct}$ | 1 | 1 | 0 | 0 |
| $D_{hex}$ | = | $13_{dec}$ | = | $15_{oct}$ | 1 | 1 | 0 | 1 |
| $E_{hex}$ | = | $14_{dec}$ | = | $16_{oct}$ | 1 | 1 | 1 | 0 |
| $F_{hex}$ | = | $15_{dec}$ | = | $17_{oct}$ | 1 | 1 | 1 | 1 |

# First Course in Python:

# Python Decimal

# Floating-Point Numbers

Python built-in class float performs some calculations that might amaze us. We all know that the sum of 1.1 and 2.2 is 3.3, but Python seems to disagree.

```
(1.1 + 2.2) == 3.3
```

```
False
```

It turns out that floating-point numbers are implemented in computer hardware as binary fractions, as computer only understands binary (0 and 1). Due to this reason, most of the decimal fractions we know, cannot be accurately stored in our computer.

```
1.1 + 2.2
```

```
Out[50]: 3.3000000000000003
```

# Decimal Module

To overcome this issue, we can use decimal module that comes with Python. While floating point numbers have precision up to 15 decimal places, the decimal module has user settable precision.

```
import decimal

print(0.1) # Output: 0.1

print(decimal.Decimal(0.1))

# Output:
Decimal('0.100000000000000005551115123125782702118158340
4541015625')
```

# Decimal Module

This module is used when we want to carry out decimal calculations like we learned in school. It also preserves significance. We know 25.50 kg is more accurate than 25.5 kg as it has two significant decimal places compared to one.

```
from decimal import Decimal as D

print(D('1.1') + D('2.2')) # Output: Decimal('3.3')


print(D('1.2') * D('2.50')) # Output: Decimal('3.000')
```

# First Course in Python:

# Python Fractions

# Fraction Module

Python provides operations involving fractional numbers through its fractions module. A fraction has a numerator and a denominator, both of which are integers. This module has support for rational number arithmetic. We can create Fraction objects in various ways.

```
import fractions

print(fractions.Fraction(1.5)) # Output: 3/2


print(fractions.Fraction(5)) # Output: 5


print(fractions.Fraction(1,3)) # Output: 1/3
```

# Fraction Module

```
import fractions


# As float
# Output: 2476979795053773/2251799813685248
print(fractions.Fraction(1.1))


# As string
# Output: 11/10
print(fractions.Fraction('1.1'))
```

# Fraction Module

This datatype supports all basic operations. Here are few examples.

```
from fractions import Fraction as F


print(F(1,3) + F(1,3)) # Output: 2/3


print(1 / F(5,6)) # Output: 6/5
```

# First Course in Python:

# Boolean

# True and False

A Boolean literal can have any of the two values: True or False.

```
x = (1 == True) # True and False are both case-sensitive
y = (1 == False)
a = True + 4
b = False + 10
print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

# Bool

We can use the bool() method to check the Boolean value of an object, which will be False for integer zero and for objects (numerical and other data types) that are empty, and True for anything else.

```
>>> bool(0)

False

>>> bool(1)

True

>>> bool(-1908)

True

>>> bool("Hello!")
```

# Practice Question

Please give the results of following operations

```
type(TURE)
type(true)
type(True)
type('True')
```

# Practice Question

Please give the results of following operations

```
True + 1
True + 1.5
True/1
True * 1
True + False
True == 1
True is 1
False == 0
False is 0
```

# Practice Question

Please give the results of following operations

```
type( True + 1)
type( True + 1.5)
type( True/1)
type( True * 1)
type( True + False)
type( True == 1)
type( True is 1)
type( False == 0)
type( False is 0)
```

# First Course in Python:

# Data Conversion

# Data Conversion

Sometimes it is necessary to convert values from one type to another. Python provides a few simple functions that will allow us to do that. The functions int, float and str will (attempt to) convert their arguments into types int, float and str respectively. We call these type conversion functions.

# Data Conversion

| Function | Description |
|---|---|
| **int(x [,base])** | Converts x to an integer. base specifies the base if x is a string. |
| **long(x [,base] )** | Converts x to a long integer. base specifies the base if x is a string. |
| **float(x)** | Converts x to a floating-point number. |
| **complex(real [,imag])** | Creates a complex number. |
| **str(x)** | Converts object x to a string representation. |
| **repr(x)** | Converts object x to an expression string. |
| **eval(str)** | Evaluates a string and returns an object. |
| **tuple(s)** | Converts s to a tuple. |
| **list(s)** | Converts s to a list. |

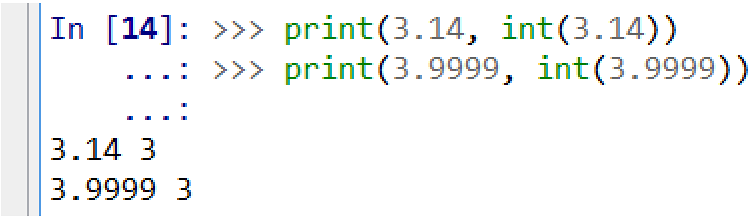
# Data Conversion

| Function | Description |
|---|---|
| **set(s)** | Converts s to a set. |
| **dict(d)** | Creates a dictionary. d must be a sequence of (key,value) tuples. |
| **frozenset(s)** | Converts s to a frozen set. |
| **chr(x)** | Converts an integer to a character. |
| **unichr(x)** | Converts an integer to a Unicode character. |
| **ord(x)** | Converts a single character to its integer value. |
| **hex(x)** | Converts an integer to a hexadecimal string. |
| **oct(x)** | Converts an integer to an octal string. |

# Data Conversion, int

The int function can take a floating point number or a string, and turn it into an int. For floating point numbers, it discards the decimal portion of the number - a process we call truncation towards zero on the number line. Let us see this in action:

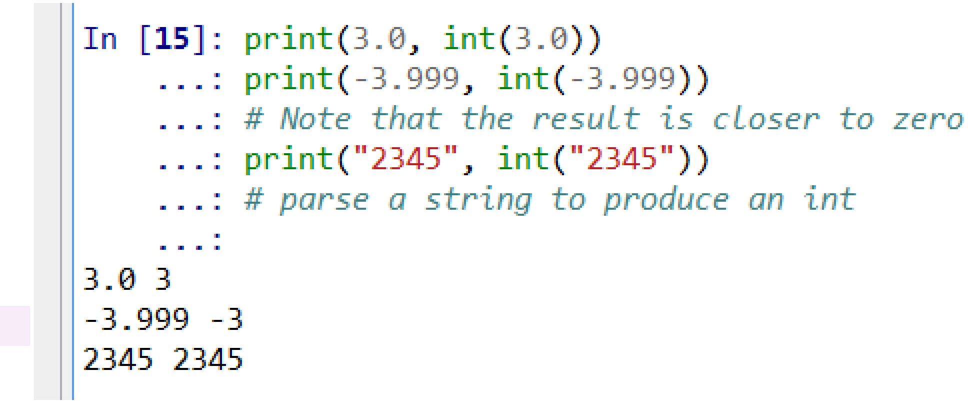
```
>>> print(3.14, int(3.14))
>>> print(3.9999, int(3.9999))
# This doesn't round to the closest int!
```



```
In [14]: >>> print(3.14, int(3.14))
    ...: >>> print(3.9999, int(3.9999))
    ...:
3.14 3
3.9999 3
```

# Data Conversion, int

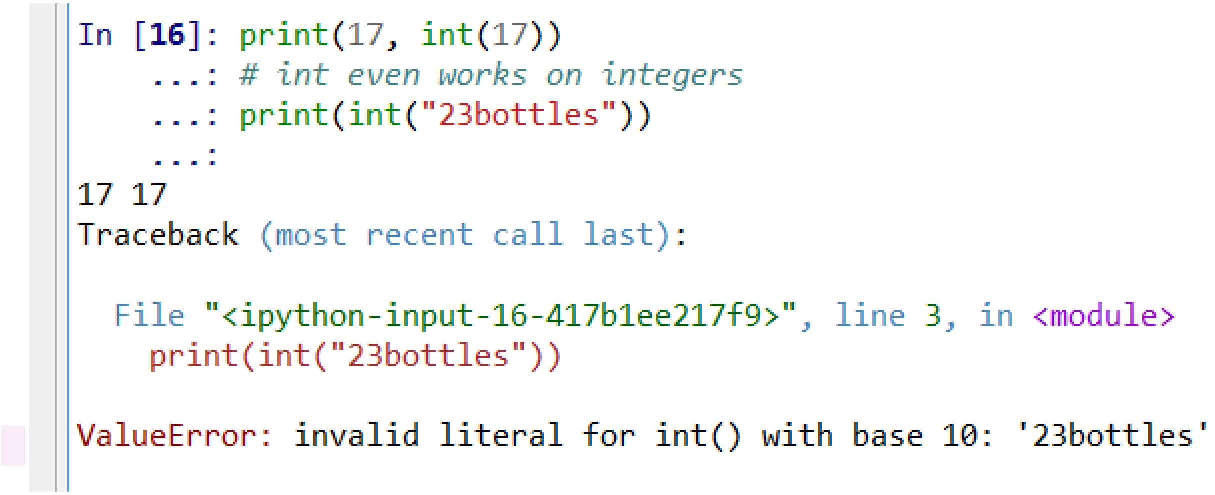
```
print(3.0, int(3.0))
print(-3.999, int(-3.999))
# Note that the result is closer to zero
print("2345", int("2345"))
# parse a string to produce an int
```



```
In [15]: print(3.0, int(3.0))
    ...: print(-3.999, int(-3.999))
    ...: # Note that the result is closer to zero
    ...: print("2345", int("2345"))
    ...: # parse a string to produce an int
    ...:
3.0 3
-3.999 -3
2345 2345
```

# Data Conversion, `int`

```
print(17, int(17))
# int even works on integers
print(int("23bottles"))
```

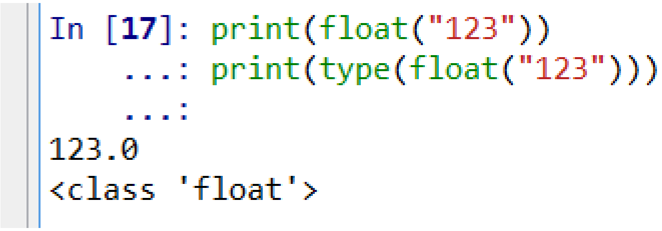


```
In [16]: print(17, int(17))
    ...: # int even works on integers
    ...: print(int("23bottles"))
    ...:
17 17
Traceback (most recent call last):

  File "<ipython-input-16-417b1ee217f9>", line 3, in <module>
    print(int("23bottles"))

ValueError: invalid literal for int() with base 10: '23bottles'
```

# Data Conversion, `float`

The type converter float can turn an integer, a float, or a syntactically legal string into a float.

```
print(float("123"))

print(type(float("123")))
```

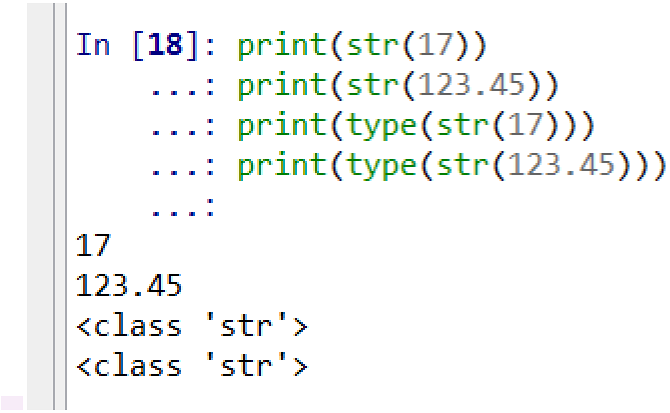


```
In [17]: print(float("123"))
    ...: print(type(float("123")))
    ...:
123.0
<class 'float'>
```

# Data Conversion, `str`

The type converter str turns its argument into a string. Remember that when we print a string, the quotes are removed. However, if we print the type, we can see that it is definitely str.

```
print(str(17))

print(str(123.45))

print(type(str(17)))

print(type(str(123.45)))
```



```
In [18]: print(str(17))
    ...: print(str(123.45))
    ...: print(type(str(17)))
    ...: print(type(str(123.45)))
    ...:
17
123.45
<class 'str'>
<class 'str'>
```

# First Course in Python:

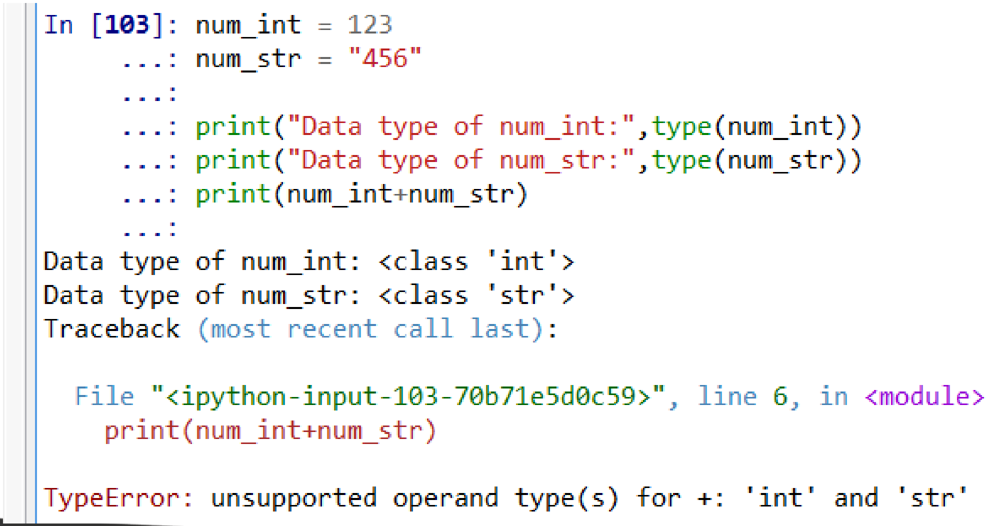
# Implicit Type Conversion

# Implicit Conversion

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo


print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

# Implicit Conversion

```
num_int = 123
num_str = "456"


print("Data type of num_int:",type(num_int))
print("Data type of num_str:",type(num_str))
print(num_int+num_str)
```



```
In [103]: num_int = 123
     ...: num_str = "456"
     ...:
     ...: print("Data type of num_int:",type(num_int))
     ...: print("Data type of num_str:",type(num_str))
     ...: print(num_int+num_str)
     ...:
Data type of num_int: <class 'int'>
Data type of num_str: <class 'str'>
Traceback (most recent call last):

  File "<ipython-input-103-70b71e5d0c59>", line 6, in <module>
    print(num_int+num_str)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Implicit Conversion

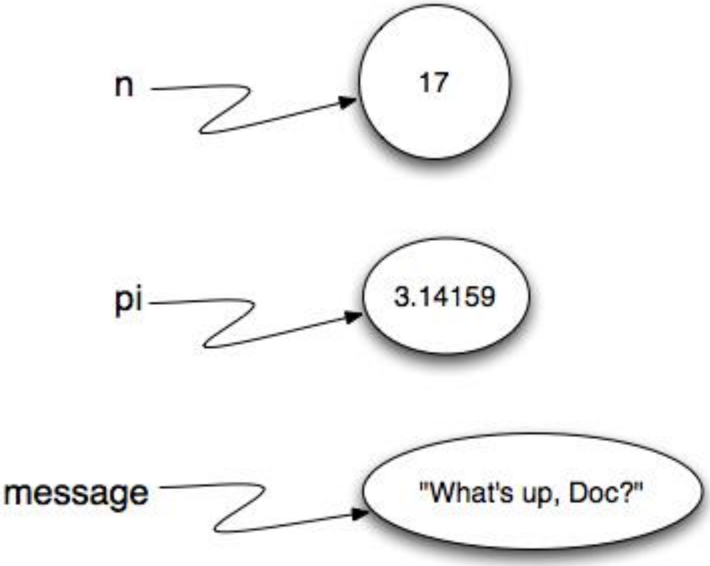
```
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))
num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))
num_sum = num_int + num_str
print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

# First Course in Python:
# Variables

# Variables

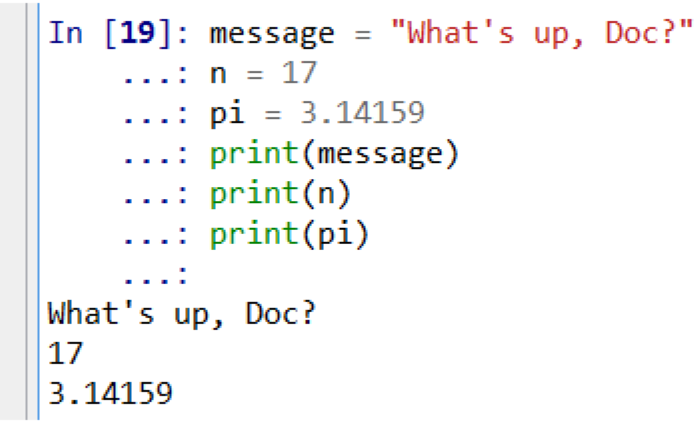
One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

Assignment statements create new variables and also give them values to refer to.



# Information Stored in Variables

```
message = "What's up, Doc?"
n = 17
pi = 3.14159
print(message)
print(n)
print(pi)
```



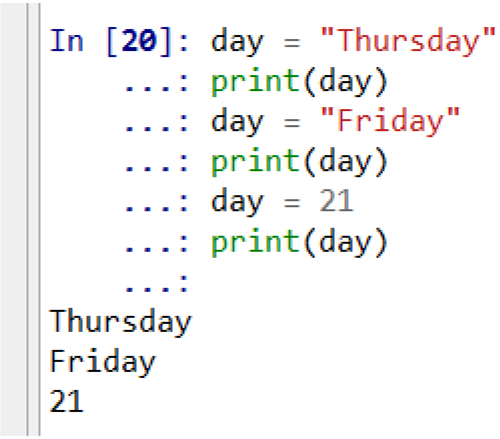
```
In [19]: message = "What's up, Doc?"
    ...: n = 17
    ...: pi = 3.14159
    ...: print(message)
    ...: print(n)
    ...: print(pi)
    ...:
What's up, Doc?
17
3.14159
```

# First Course in Python

We use variables in a program to “remember” things, like the current score at the football game. But variables are variable. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable.

# Update the info in a variable

```
day = "Thursday"

print(day)

day = "Friday"

print(day)

day = 21

print(day)
```



```
In [20]: day = "Thursday"
    ...: print(day)
    ...: day = "Friday"
    ...: print(day)
    ...: day = 21
    ...: print(day)
    ...:
Thursday
Friday
21
```

# Variable Names

Variable names can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore.

Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables.

Caution: Variable names can never contain spaces.

# Variable Names

- Rules
  - Can not contain spaces
  - Are case sensitive

    `firstName` and `firstname` are two different variables
  - Cannot start with a number
  - Cannot use special symbols like !, @, #, $, %
- Guidelines
  - Should be descriptive but not too long (`favoriteSign` not `yourFavoriteSignInTheHoroscope`)
  - Use a casing "scheme"

    `camelCasing` or `PascalCasing` or `Use_underscore`

# Use Underscore

The underscore character _ can also appear in a name. It is often used in names with multiple words, such as

`my_name` or

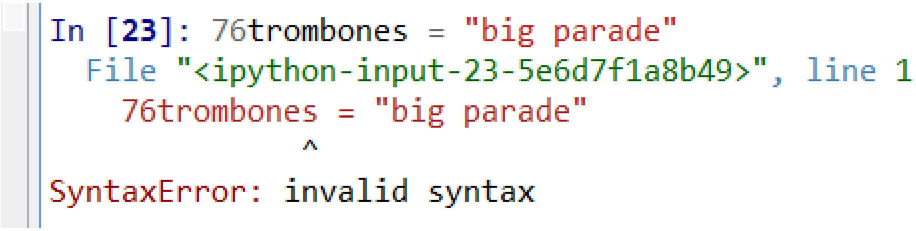
`price_of_tea_in_china`

There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.

# First Course in Python

If you give a variable an illegal name, you get a syntax error. In the example below, each of the variable names is illegal.

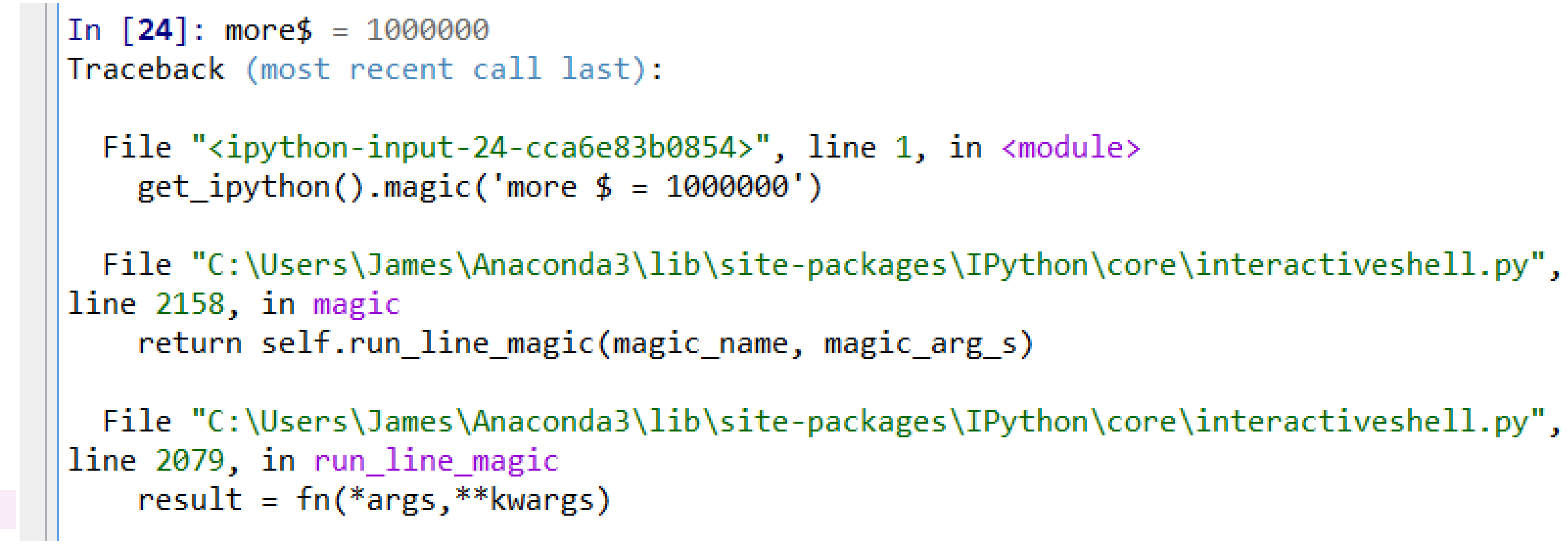
```
76trombones = "big parade"
```



```
In [23]: 76trombones = "big parade"
  File "<ipython-input-23-5e6d7f1a8b49>", line 1
    76trombones = "big parade"
              ^
SyntaxError: invalid syntax
```

# First Course in Python

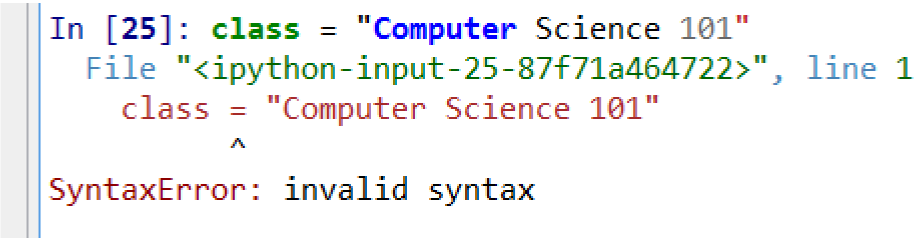
```
more$ = 1000000
```



```
In [24]: more$ = 1000000
Traceback (most recent call last):

  File "<ipython-input-24-cca6e83b0854>", line 1, in <module>
    get_ipython().magic('more $ = 1000000')

  File "C:\Users\James\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py",
line 2158, in magic
    return self.run_line_magic(magic_name, magic_arg_s)

  File "C:\Users\James\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py",
line 2079, in run_line_magic
    result = fn(*args,**kwargs)
```

# First Course in Python

```
class = "Computer Science 101"
```



```
In [25]: class = "Computer Science 101"
  File "<ipython-input-25-87f71a464722>", line 1
    class = "Computer Science 101"
          ^
SyntaxError: invalid syntax
```

# Python Keywords

Python 3 has these keywords:

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

# Special literals

Python contains one special literal i.e. None. We use it to specify to that field that is not created.

```
drink = "Available"
food = None
def menu(x):
    if x == drink:
        print(drink)
    else:
        print(food)
menu(drink)
menu(food)
```

# First Course in Python:

# Info Input and Output

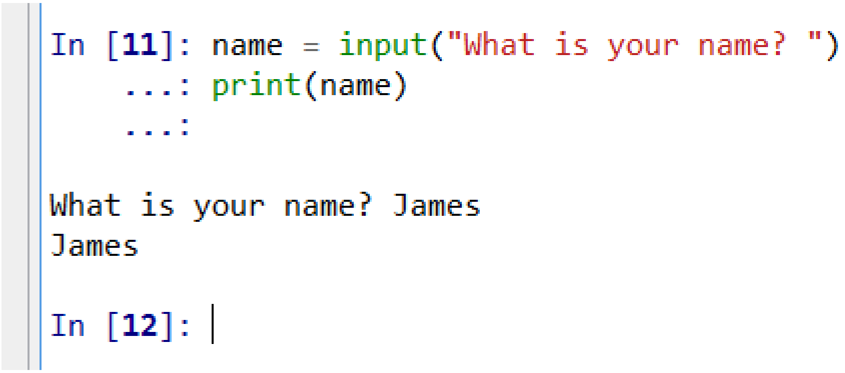
# ask a user for information

The input function allows you to specify a message to display and returns the value typed in by the user.

We use a variable to remember the value entered by the user.

We called our variable “name” but you can call it just about anything as long the variable name doesn’t contain spaces

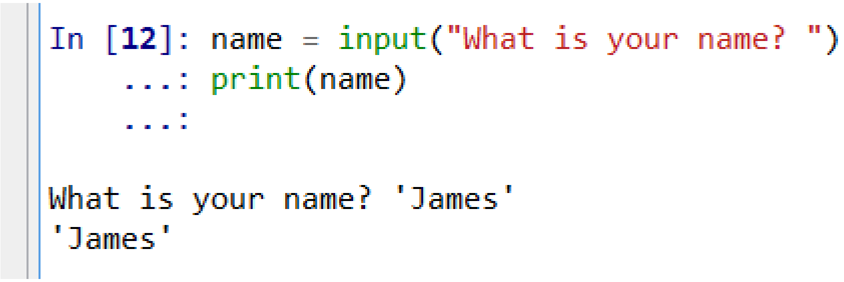
```
name = input("What is your name? ")
print(name)
```



```
In [11]: name = input("What is your name? ")
    ...: print(name)
    ...:

What is your name? James
James

In [12]:
```

# Input

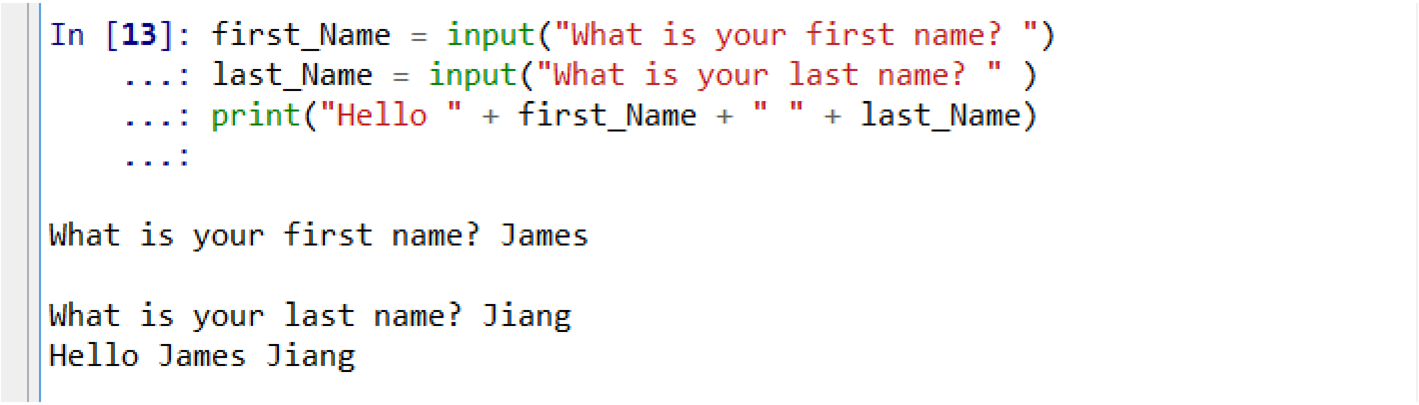
```
name = input("What is your name? ")
print(name)
```



```
In [12]: name = input("What is your name? ")
    ...: print(name)
    ...:

What is your name? 'James'
'James'
```

# Input

```
first_Name = input("What is your first name? ")
last_Name = input("What is your last name? " )
print("Hello " + first_Name + " " + last_Name)
```



```
In [13]: first_Name = input("What is your first name? ")
    ...: last_Name = input("What is your last name? " )
    ...: print("Hello " + first_Name + " " + last_Name)
    ...:

What is your first name? James

What is your last name? Jiang
Hello James Jiang
```

# Deep Dive into Print

```
a = 5

print('The value of a is', a)

# Output: The value of a is 5
```

In the second print() statement, we can notice that a space was added between the string and the value of variable a.This is by default, but we can change it.

The actual syntax of the print() function is

```
print(*objects, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

# Deep Dive into Print

```
print(*objects, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

Here, objects is the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen). Here are an example to illustrate this.

# Deep Dive into Print

```
print(1,2,3,4)

# Output: 1 2 3 4


print(1,2,3,4,sep='*')

# Output: 1*2*3*4


print(1,2,3,4,sep='#',end='&')

# Output: 1#2#3#4&
```

# Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is
{}'.format(x,y))
The value of x is 5 and y is 10
```

Here the curly braces {} are used as placeholders.

# Output formatting

We can specify the order in which it is printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread','butter'))

# Output: I love bread and butter


print('I love {1} and {0}'.format('bread','butter'))

# Output: I love butter and bread
```

# Output formatting

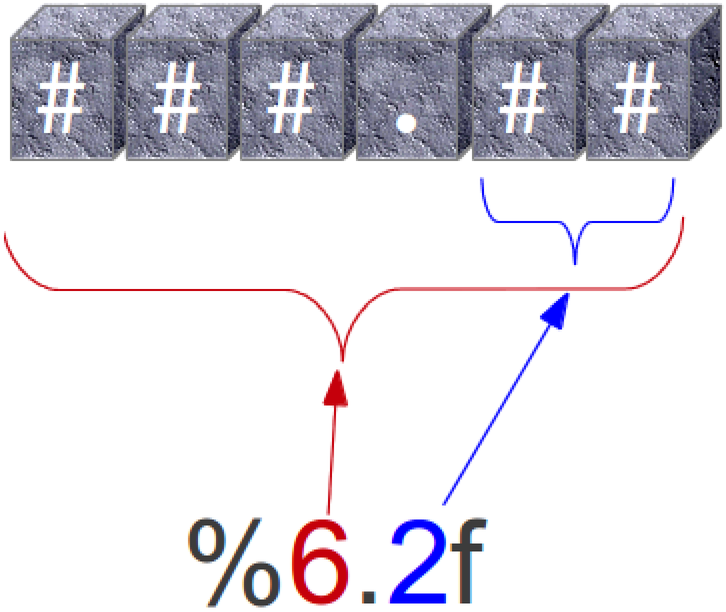
We can even use keyword arguments to format the string.

```
>>> print('Hello {name}, {greeting}'.format(greeting =
'Good morning', name = 'John'))
```

```
Hello John, Good morning
```

# Formatting

```
x = 12.3456789

>>> print('The value of x is %3.2f' %x)

>>> print('The value of x is %3.4f' %x)
```



Strings:

# About Strings

# About Strings

Python has a built-in string class named "str" with many handy features (there is an older module named "string" which you should not use).

String literals can be enclosed by either double or single quotes, although single quotes are more commonly used.

Backslash escapes work the usual way within both single and double quoted literals -- e.g. \n \' \".

A double quoted string literal can contain single quotes without any fuss (e.g. "I didn't do it") and likewise single quoted string can contain double quotes.

A string literal can span multiple lines, but there must be a backslash \ at the end of each line to escape the newline.

String literals inside triple quotes, """" or ''', can span multiple lines of text.

# About Strings

Python strings are "immutable" which means they cannot be changed after they are created.

Since strings can't be changed, we construct *new* strings as we go to represent computed values.

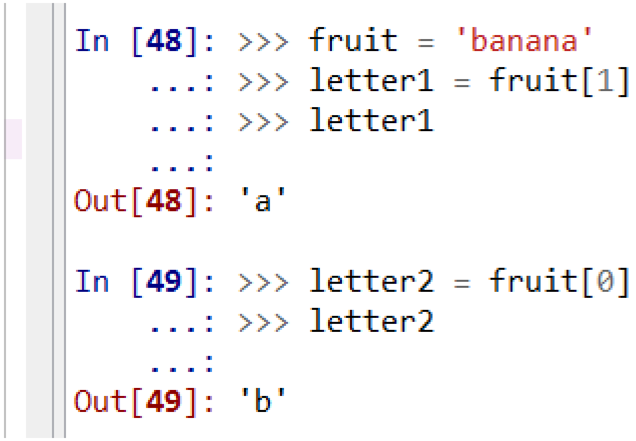
So for example the expression ('hello' + 'there') takes in the 2 strings 'hello' and 'there' and builds a new string 'hellothere’.

Characters in a string can be accessed using the standard [ ] syntax, and like Java and C++, Python uses zero-based indexing, so if str is 'hello' str[1] is 'e'. If the index is out of bounds for the string, Python raises an error.

# Strings:
# Basic Operations

# A String Is a Sequence

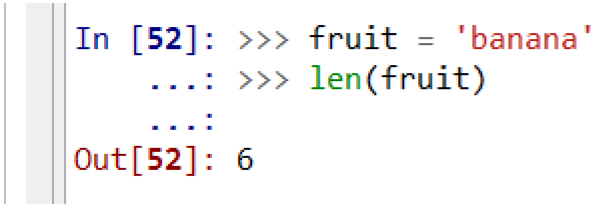
```
>>> fruit = 'banana'
>>> letter1 = fruit[1]
>>> letter1
>>> letter2 = fruit[0]
>>> letter2
```



```
In [48]: >>> fruit = 'banana'
    ...: >>> letter1 = fruit[1]
    ...: >>> letter1
    ...:
Out[48]: 'a'

In [49]: >>> letter2 = fruit[0]
    ...: >>> letter2
    ...:
Out[49]: 'b'
```

# Length of a Sequence

```
>>> fruit = 'banana'

>>> len(fruit)
```



```
In [52]: >>> fruit = 'banana'
    ...: >>> len(fruit)
    ...:
Out[52]: 6
```

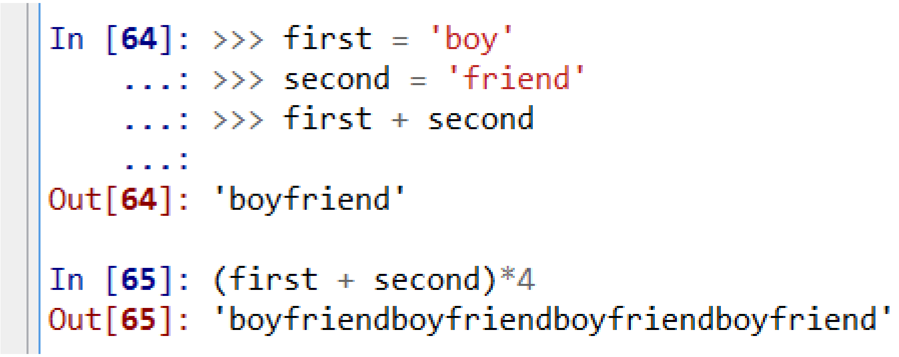
# Items in Strings Are Immutable

```
>>> fruit[0] = 'J'

TypeError: 'str' object does not support item assignment
```

# string concatenation

The + operator performs string concatenation, which means it joins the strings by linking them end-to-end. For example:

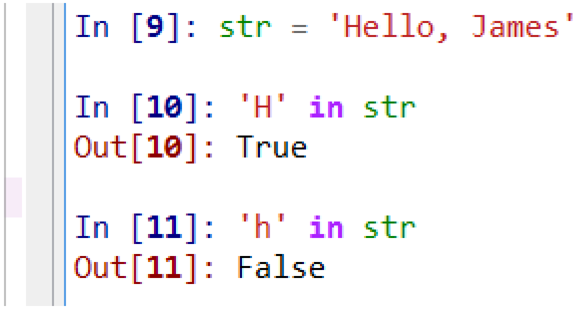
```
>>> first = 'boy'

>>> second = 'friend'

>>> first + second

>>> (first + second)*4
```



```
In [64]: >>> first = 'boy'
    ...: >>> second = 'friend'
    ...: >>> first + second
    ...:
Out[64]: 'boyfriend'

In [65]: (first + second)*4
Out[65]: 'boyfriendboyfriendboyfriendboyfriend'
```

# Membership

```
str = 'Hello, James'
'H' in str
'h' in str
```



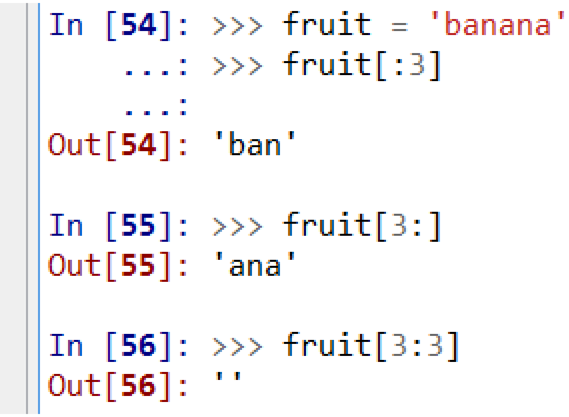
```
In [9]: str = 'Hello, James'

In [10]: 'H' in str
Out[10]: True

In [11]: 'h' in str
Out[11]: False
```

# Strings:
# Slices

# Positive Index Number

| b | a | n | a | n | a |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

```
>>> fruit = 'banana'

>>> fruit[:3]

>>> fruit[3:]

>>> fruit[3:3]
```

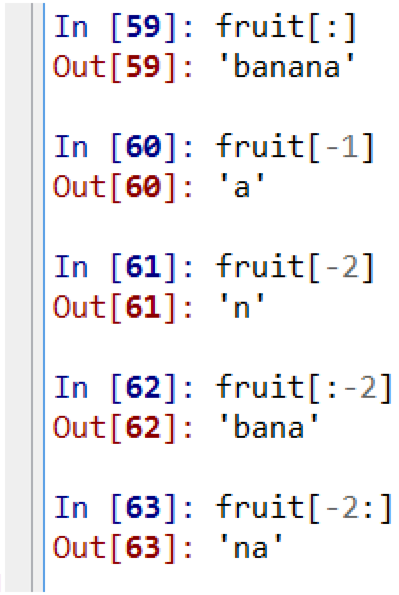


```
In [54]: >>> fruit = 'banana'
    ...: >>> fruit[:3]
    ...:
Out[54]: 'ban'

In [55]: >>> fruit[3:]
Out[55]: 'ana'

In [56]: >>> fruit[3:3]
Out[56]: ''
```

# Non-Positive Index Number

| b | a | n | a | n | a |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

```
>>> fruit[:]

>>> fruit[-1]

>>> fruit[-2]

>>> fruit[:-2]

>>> fruit[-2:]
```

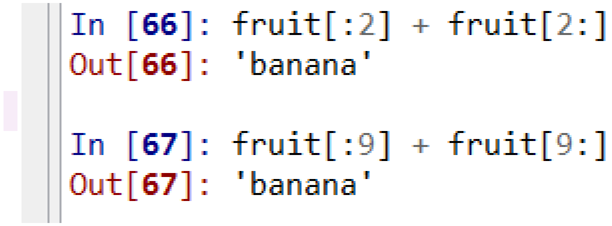


```
In [59]: fruit[:]
Out[59]: 'banana'

In [60]: fruit[-1]
Out[60]: 'a'

In [61]: fruit[-2]
Out[61]: 'n'

In [62]: fruit[:-2]
Out[62]: 'bana'

In [63]: fruit[-2:]
Out[63]: 'na'
```

# Add Slices

| | | | | | |
|---|---|---|---|---|---|
| b | a | n | a | n | a |
| 0 | 1 | 2 | 3 | 4 | 5 |

```
>>> fruit[:2] + fruit[2:]
>>> fruit[:9] + fruit[9:]
```

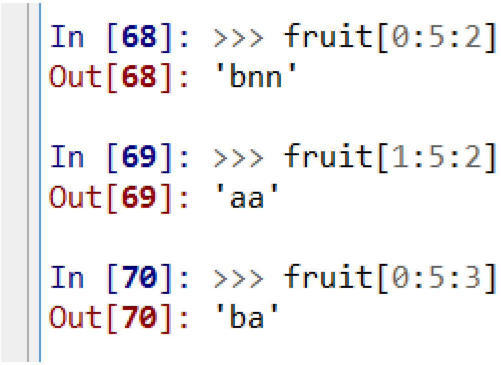


```
In [66]: fruit[:2] + fruit[2:]
Out[66]: 'banana'

In [67]: fruit[:9] + fruit[9:]
Out[67]: 'banana'
```

# Stride

| | | | | | |
|---|---|---|---|---|---|
| b | a | n | a | n | a |
| 0 | 1 | 2 | 3 | 4 | 5 |

```
>>> fruit[0:5:2]

>>> fruit[1:5:2]

>>> fruit[0:5:3]
```

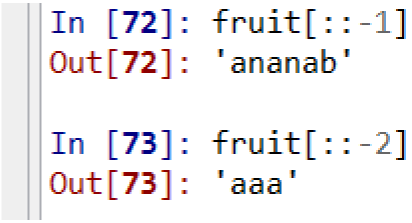


```
In [68]: >>> fruit[0:5:2]
Out[68]: 'bnn'

In [69]: >>> fruit[1:5:2]
Out[69]: 'aa'

In [70]: >>> fruit[0:5:3]
Out[70]: 'ba'
```

# Stride

| | | | | | |
|---|---|---|---|---|---|
| b | a | n | a | n | a |
| 0 | 1 | 2 | 3 | 4 | 5 |

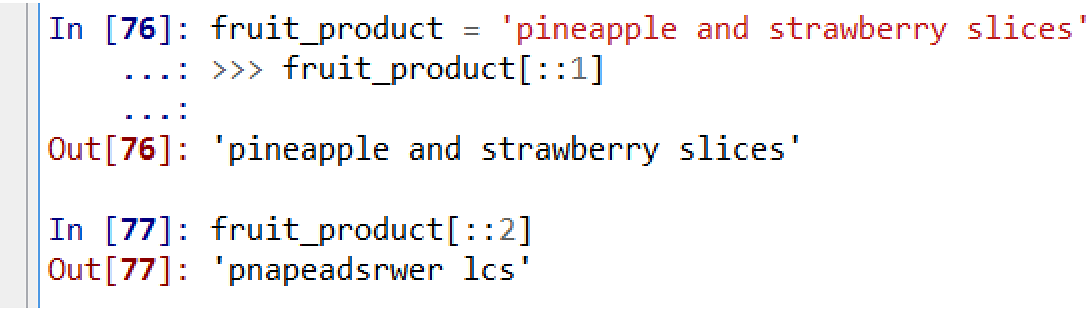
```
>>> fruit[::-1]
>>> fruit[::-2]
```



```
In [72]: fruit[::-1]
Out[72]: 'ananab'

In [73]: fruit[::-2]
Out[73]: 'aaa'
```

# Strides

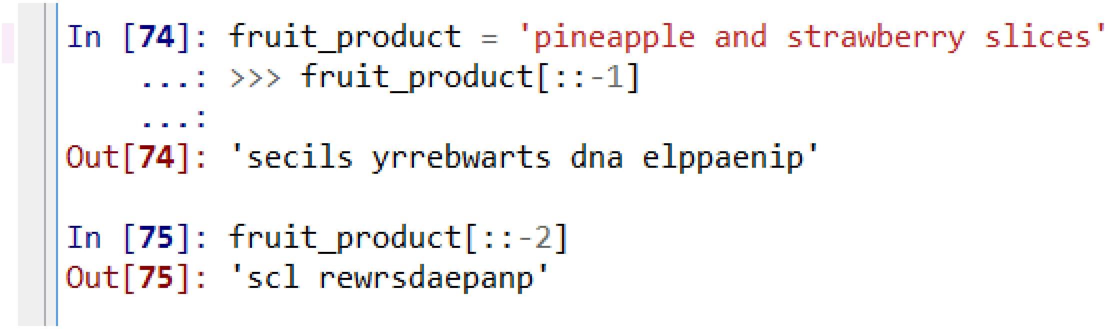
```
fruit_product = 'pineapple and strawberry slices'
>>> fruit_product[::1]
>>> fruit_product[::2]
```



```
In [76]: fruit_product = 'pineapple and strawberry slices'
    ...: >>> fruit_product[::1]
    ...:
Out[76]: 'pineapple and strawberry slices'

In [77]: fruit_product[::2]
Out[77]: 'pnapeadsrwer lcs'
```

# Strides

```
fruit_product = 'pineapple and strawberry slices'
>>> fruit_product[::-1]
>>> fruit_product[::-2]
```



```
In [74]: fruit_product = 'pineapple and strawberry slices'
    ...: >>> fruit_product[::-1]
    ...:
Out[74]: 'secils yrrebwarts dna elppaenip'

In [75]: fruit_product[::-2]
Out[75]: 'scl rewrsdaepanp'
```

## Strings:

# String Formatting Operator

# Formatting Operators

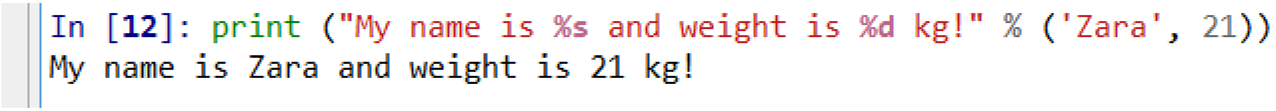
| Format Symbol | Conversion |
|---|---|
| `%c` | character |
| `%s` | string conversion via str() prior to formatting |
| `%i` | signed decimal integer |
| `%d` | signed decimal integer |
| `%u` | unsigned decimal integer |
| `%o` | octal integer |
| `%x` | hexadecimal integer (lowercase letters) |

# Formatting Operators

| Format Symbol | Conversion |
|---|---|
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e') |
| %E | exponential notation (with UPPERcase 'E') |
| %f | floating point real number |
| %g | the shorter of %f and %e |
| %G | the shorter of %f and %E |

# Formatting Operators

```
print ("My name is %s and weight is %d kg!" %
('Zara', 21))
```



```
In [12]: print ("My name is %s and weight is %d kg!" % ('Zara', 21))
My name is Zara and weight is 21 kg!
```

# Formatting Operations

For example, suppose your program wants to report how many bananas you have, and you have an int variable named nBananas that contains the actual banana count, and you want to print a string something like “We have 27 bananas” if nBananas has the value 27. This is how you do it:

```
nBananas = 27

"We have %d bananas." % nBananas
```

# Formatting Operations

In general, when a string value appears on the left side of the “%” operator, that string is called the format string. Within a format string, the percent character “%” has special meaning. In the example above, the “%d” part means that an integer value will be substituted into the format string at that position. So the result of the format operator will be a string containing all the characters from the format string, except that the value on the right of the operator (27) will replace the “%d” in the format string.

# Formatting Operations

```
nBananas = 27
"We have %6d bananas." % nBananas


nBananas = 27
"We have %8d bananas." % nBananas


caseCount = 42
caseContents = "peaches"
print "We have %d cases of %s today." % (caseCount,
caseContents)
```

# Formatting Operations

```
'%s' % 'soup' # default: aligned right

'%6s' % 'soup'

'%-10s' % 'soup' # aligned left

"%d" % 1107

"%5d" % 1107

'%30d' % 1107

'%2d' % 1107

'%5d' % 505

'%-5d' % 505

'%05d'%42
```

# Formatting Operations

```
"%f" % 0.0

"%f" % 1.5

pi = 3.141592653589793

"%f" % pi

"%.0f" % pi

"%.15f" % pi

"%5.1f" % pi

"%5.3f" % pi
```

# Strings:

# Formatters with Placeholders

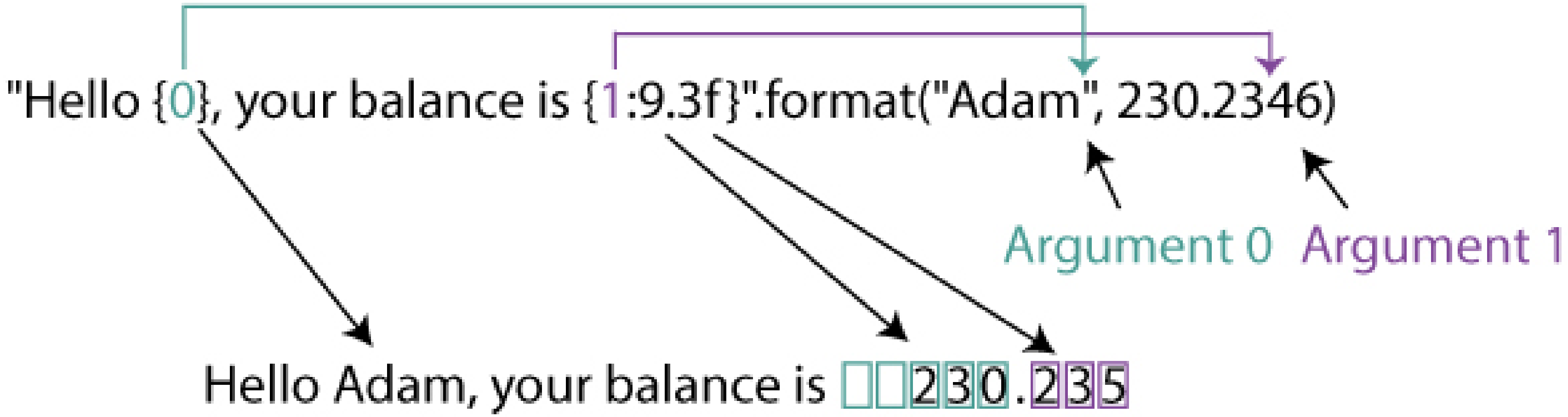
# .format

Formatters work by putting in one or more replacement fields or placeholders — defined by a pair of curly braces {} — into a string and calling the str.format() method. You'll pass into the method the value you want to concatenate with the string. This value will be passed through in the same place that your placeholder is positioned when you run the program.

```
print("Sammy has {} balloons.".format(5))
```

# .format

The format() reads the type of arguments passed to it and formats it according to the format codes defined in the string.


"Hello {0}, your balance is {1:9.3f}".format("Adam", 230.2346)
Argument 0
Argument 1
Hello Adam, your balance is □□230.235

# .format

```
new_open_string = "Sammy loves {} {}."
#2 {} placeholders

print(new_open_string.format("open-source",
"software"))

#Pass 2 strings into method, separated by a comma
```

# .format

```
sammy_string = "Sammy loves {} {}, and has {} {}."
#4 {} placeholders

print(sammy_string.format("open-source",
"software", 5, "balloons"))

#Pass 4 strings into method
```

# .format

We can pass these index numbers into the curly braces that serve as the placeholders in the original string:

```
print("Sammy the {0} has a pet
{1}!".format("shark", "pilot fish"))
```

```
print("Sammy the {1} has a pet
{0}!".format("shark", "pilot fish"))
```

```
print("Sammy the {2} has a pet
{1}!".format("shark", "pilot fish")) # Error
```

# .format

Let’s look at an example where we have an integer passed through the method, but want to display it as a float by adding the f conversion type argument:

```
print("Sammy ate {0:f} percent of a
{1}!".format(75.12345678, "pizza")) #default: 6 digits
```

```
print("Sammy ate {0:.2f} percent of a
{1}!".format(75.12345678, "pizza"))
```

# .format

```
print("Sammy ate {0:.4f} percent of a
{1}!".format(75.12345678, "pizza"))
```

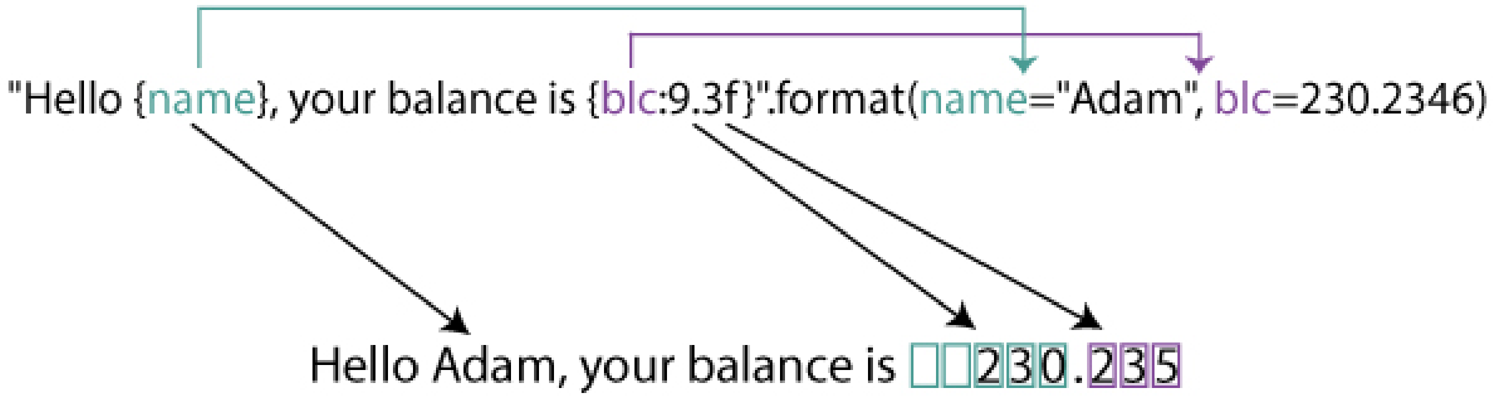
```
print("Sammy ate {0:.5f} percent of a
{1}!".format(75.12345678, "pizza"))
```

```
print("Sammy ate {0:.0f} percent of a
{1}!".format(75.12345678, "pizza"))
```

# .format

```
print("Sammy ate {0:10.4f} percent of a
{1}!".format(75.12345678, "pizza"))
```

```
print("Sammy ate {0:20.0f} percent of a
{1}!".format(75.12345678, "pizza"))

# allotted a minimum of 20 places including the "."
```



"Hello {name}, your balance is {blc:9.3f}".format(name="Adam", blc=230.2346)

Hello Adam, your balance is □□230.235

# .format

```
print("Sammy has {0:4} red {1:16}!".format(5,
"balloons"))
```

```
print("Sammy has {0:2} red {1:10}!".format(5,
"balloons"))
```

```
print("Sammy has {0:1} red {1:20}!".format(5,
"balloons"))
```

# .format

You can modify this by placing an alignment code just following the colon. < will left-align the text in a field, ^ will center the text in the field, and > will right-align it.

```
print("Sammy has {0:<4} red {1:^16}!".format(5,
"balloons"))
```

```
print("Sammy has {0:^4} red {1:>16}!".format(5,
"balloons"))
```

# .format

By default, when we make a field larger with formatters, Python will fill the field with whitespace characters. We can modify that to be a different character by specifying the character we want it to be directly following the colon:

```
print("Sammy has {0:*<4} red {1:@^16}!".format(5,
"balloons"))
```

```
print("Sammy has {0:$^4} red {1:!>16}!".format(5,
"balloons"))
```

# Using Formatters to Organize Data

Formatters can be seen in their best light when they are being used to organize a lot of data in a visual way. If we are showing databases to users, using formatters to increase field size and modify alignment can make your output more readable.

Let’s look at a typical for loop in Python that will print out i, i*i, and i*i*i in the range from 3 to 12:

```
for i in range(3,13):
    print(i, i*i, i*i*i)
```

# Using Formatters to Organize Data

```
for i in range(3,13):
    print("{:3d} {:4d} {:5d}".format(i, i*i,
i*i*i))


for i in range(3,13):
    print("{:6d} {:6d} {:6d}".format(i, i*i,
i*i*i))
```

# Using Formatters to Organize Data

```
for i in range(3,13):
    print("{:^6d} {:^6d} {:^6d}".format(i, i*i, 
i*i*i))
```

```
for i in range(3,13):
    print("{:<6d} {:<6d} {:<6d}".format(i, i*i, 
i*i*i))
```

Strings:

# String Methods

# String Methods

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Here are some of the most common string methods:

- `s.lower()`, `s.upper()` -- returns the lowercase or uppercase version of the string
- `s.strip()` -- returns a string with whitespace removed from the start and end
- `s.isalpha()`/`s.isdigit()`/`s.isspace()`... -- tests if all the string chars are in the various character classes

# String Methods

- s`.startswith('other')`, s.endswith('other') -- tests if the string starts or ends with the given other string
- s`.find('other')` -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s`.replace('old', 'new')` -- returns a string where all occurrences of 'old' have been replaced by 'new'

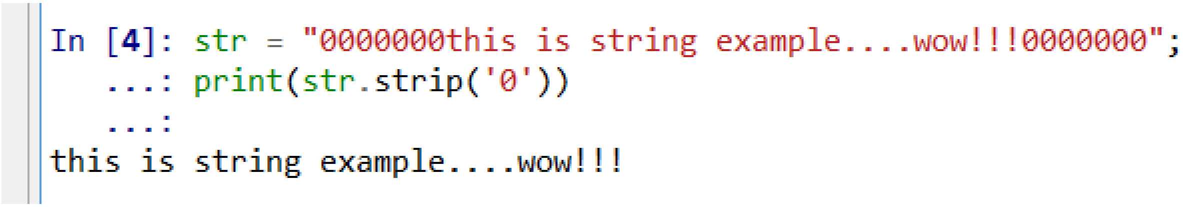
# String Methods

- s`.split('delim')` -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s`.join(list)` -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

# .strip()

The method strip() returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

```
str = "0000000this is string example....wow!!!0000000";
print(str.strip('0'))
```

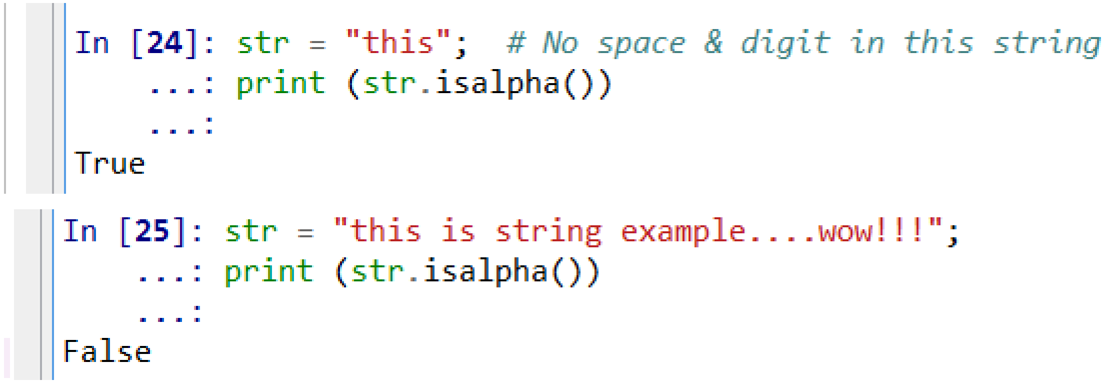


```
In [4]: str = "0000000this is string example....wow!!!0000000";
   ...: print(str.strip('0'))
   ...:
this is string example....wow!!!
```

# .islpha()

This method returns true if all characters in the string are alphabetic and there is at least one character, false otherwise.

```
str = "this";  # No space & digit in this string
print (str.isalpha())
str = "this is string example....wow!!!";
print (str.isalpha())
```

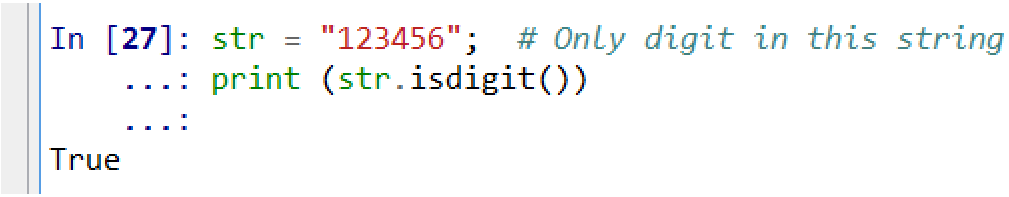


```
In [24]: str = "this";  # No space & digit in this string
    ...: print (str.isalpha())
    ...:
True

In [25]: str = "this is string example....wow!!!";
    ...: print (str.isalpha())
    ...:
False
```

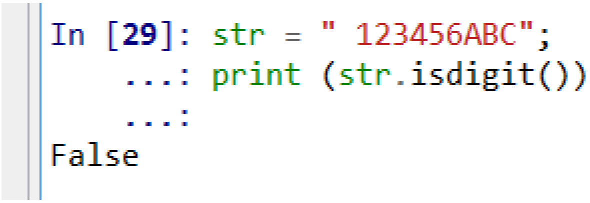
# .isdigit()

This method returns true if all characters in the string are digits and there is at least one character, false otherwise.

```
str = "123456";  # Only digit in this string
print (str.isdigit())
str = " 123456ABC";
print (str.isdigit())
```



```
In [27]: str = "123456";  # Only digit in this string
    ...: print (str.isdigit())
    ...:
True
```

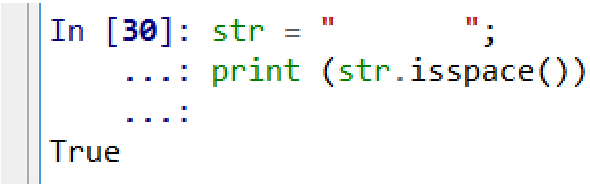


```
In [29]: str = " 123456ABC";
    ...: print (str.isdigit())
    ...:
False
```

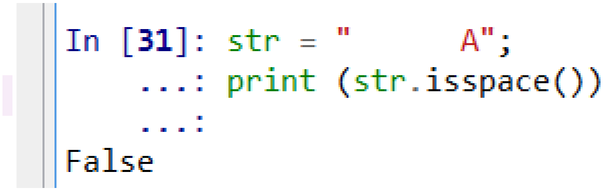
# .isspace()

This method returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

```
str = "         ";
print (str.isspace())
str = "       A";
print (str.isspace())
```



```
In [30]: str = "         ";
    ...: print (str.isspace())
    ...:
True
```



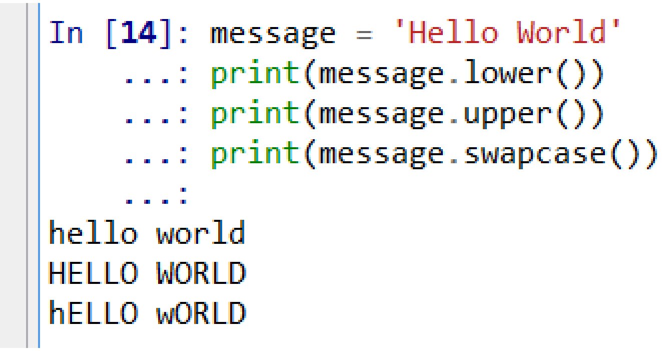
```
In [31]: str = "       A";
    ...: print (str.isspace())
    ...:
False
```

# Strings:

# Working on Letters

# Manipulate String

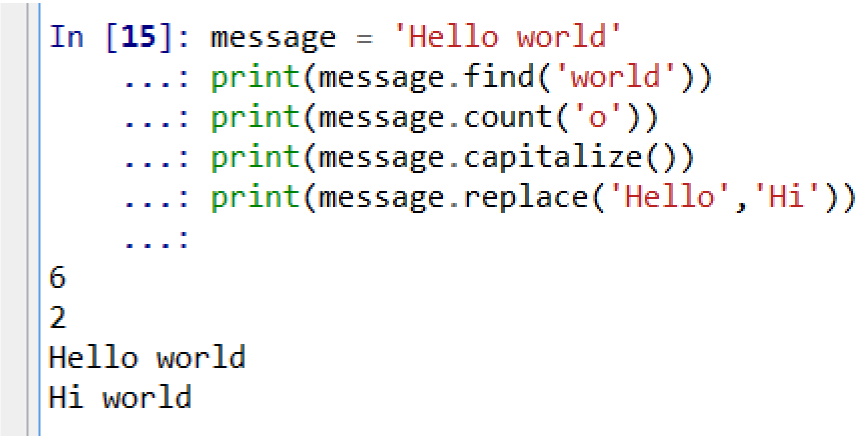
```
message = 'Hello World'
print(message.lower())
print(message.upper())
print(message.swapcase())
```



```
In [14]: message = 'Hello World'
    ...: print(message.lower())
    ...: print(message.upper())
    ...: print(message.swapcase())
    ...:
hello world
HELLO WORLD
hELLO wORLD
```

# Manipulate String

```
message = 'Hello world'
print(message.find('world'))
print(message.count('o'))
print(message.capitalize())
# It returns a copy of the string
# with only its first character capitalized.
print(message.replace('Hello','Hi'))
```

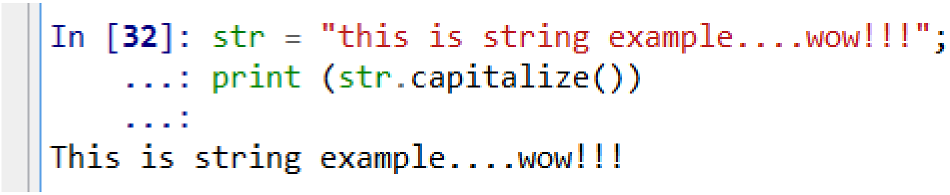


```
In [15]: message = 'Hello world'
    ...: print(message.find('world'))
    ...: print(message.count('o'))
    ...: print(message.capitalize())
    ...: print(message.replace('Hello','Hi'))
    ...:
6
2
Hello world
Hi world
```

# Capitalize

It returns a copy of the string with only its first character capitalized.

```
str = "this is string example....wow!!!";

print (str.capitalize())
```

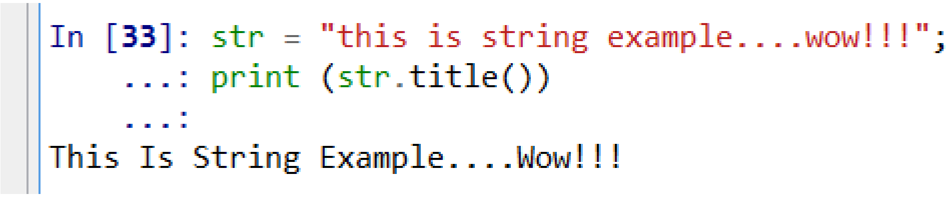


```
In [32]: str = "this is string example....wow!!!";
    ...: print (str.capitalize())
    ...:
This is string example....wow!!!
```

# Capitalize, Title

The method title() returns a copy of the string in which first characters of all the words are capitalized.

```
str = "this is string example....wow!!!";
print (str.title())
```



```
In [33]: str = "this is string example....wow!!!";
    ...: print (str.title())
    ...:
This Is String Example....Wow!!!
```

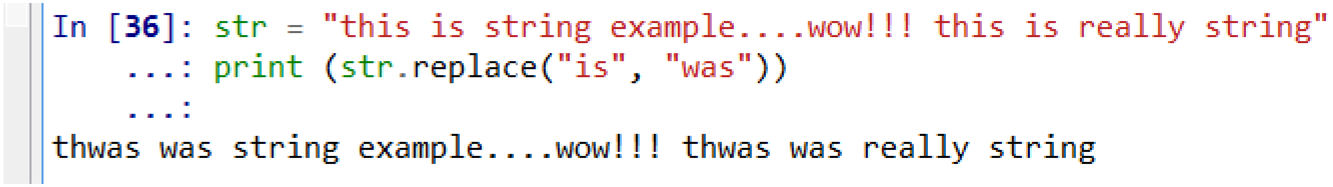
# Replace

This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

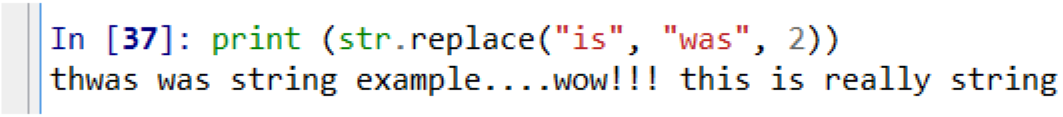
```
str = "this is string example....wow!!! this is really
string"
```

```
print (str.replace("is", "was"))
```

```
print (str.replace("is", "was", 2))
```



```
In [36]: str = "this is string example....wow!!! this is really string"
    ...: print (str.replace("is", "was"))
    ...:
thwas was string example....wow!!! thwas was really string
```



```
In [37]: print (str.replace("is", "was", 2))
thwas was string example....wow!!! this is really string
```

# Strings:

# Counting Methods

# Count

```
ss = "bananas are always good"

print(ss.count("a"))

print(ss.count("s"))

print(ss.count("q"))

print(ss.find("m"))

# check to see where the first “m” occurs in the string
ss
```

# Count and Find

```
likes = "Sammy likes to swim in the ocean, likes to spin up servers, and likes to smile."

print(likes.count("likes"))

print(likes.find("likes"))

# Instead of starting at the beginning of the string, let’s start after the index number 9:

print(likes.find("likes", 9))

# Like slicing, we can do so by counting backwards using a negative index number:

print(likes.find("likes", 40, -6))
```

# Strings:
# Format

# format() Method for Formatting Strings

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contains curly braces {} as placeholders or replacement fields which gets replaced. We can use positional arguments or keyword arguments to specify the order.

```
# default(implicit) order

default_order = "{}, {} and
{}".format('John','Bill','Sean')

print('\n--- Default Order ---')

print(default_order)
```

# format() Method for Formatting Strings

```
# order using positional argument

positional_order = "{1}, {0} and
{2}".format('John','Bill','Sean')

print('\n--- Positional Order ---')

print(positional_order)


# order using keyword argument

keyword_order = "{s}, {b} and
{j}".format(j='John',b='Bill',s='Sean')

print('\n--- Keyword Order ---')

print(keyword_order)
```

# Strings:
# Other Methods

# Iterating Through String

```
count = 0
for letter in 'Hello World':
    if(letter == 'o'):
        count += 1
print(count,'letter o found')
```

# an enumerate object

```
str = 'cold'


# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)
# list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'),
(3, 'd')]


#character count
print('len(str) = ', len(str))
```

# Escape Sequence

If we want to print a text like -He said, "What's there?"- we can neither use single quote or double quotes. This will result into SyntaxError as the text itself contains both single and double quotes.

```
print("He said, "What's there?"")

print('He said, "What's there?"')
```

# Escape Sequence

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently. If we use single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

```
print('''He said, "What's there?"''')

# escaping single quotes

print('He said, "What\'s there?"')

# escaping double quotes

print("He said, \"What's there?\"")
```

# Escape Sequence in Python

| Escape Sequence | Description |
|---|---|
| \newline | Backslash and newline ignored |
| \\ | Backslash |
| \' | Single quote |
| \" | Double quote |
| \a | ASCII Bell |
| \b | ASCII Backspace |

# Escape Sequence in Python

| Escape Sequence | Description |
|---|---|
| \f | ASCII Formfeed |
| \n | ASCII Linefeed |
| \r | ASCII Carriage Return |
| \t | ASCII Horizontal Tab |
| \v | ASCII Vertical Tab |
| \ooo | Character with octal value ooo |
| \xHH | Character with hexadecimal value HH |

# Escape Sequence

```
print("C:\\Python32\\Lib")
C:\Python32\Lib
```

```
>>> print("This is printed\nin two lines")
This is printed
in two lines
```

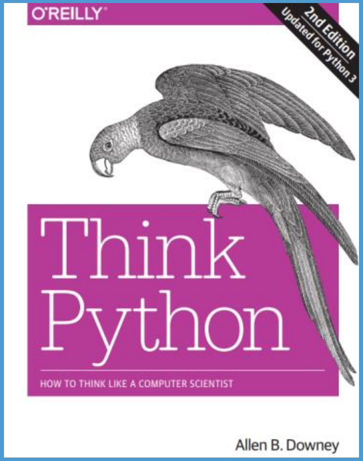
```
>>> print("This is \x48\x45\x58 representation")
This is HEX representation
```

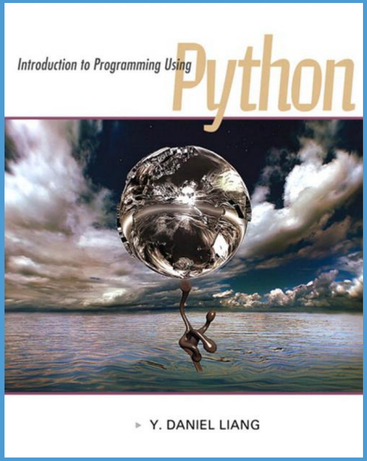
# Raw String to ignore escape sequence

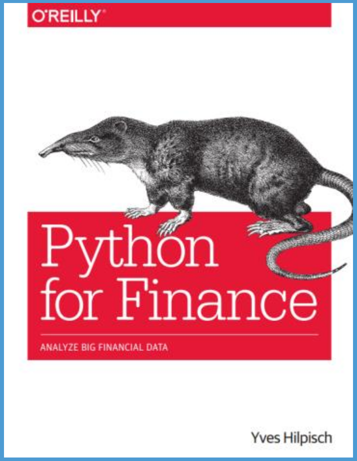
Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

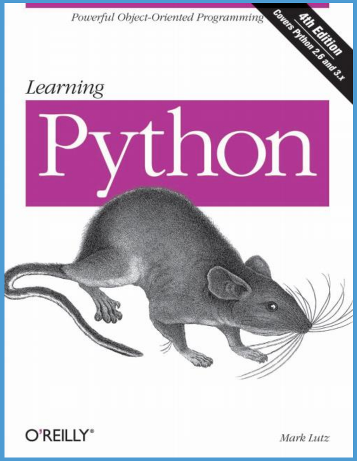
```
print("This is \x61 \ngood example")

This is a

good example

>>> print(r"This is \x61 \ngood example")

This is \x61 \ngood example
```

# References


O'REILLY
2nd Edition
Updated for Python 3
Think Python
HOW TO THINK LIKE A COMPUTER SCIENTIST
Allen B. Downey


Introduction to Programming Using Python
Y. DANIEL LIANG


O'REILLY
Python for Finance
ANALYZE BIG FINANCIAL DATA
Yves Hilpisch


Powerful Object-Oriented Programming
4th Edition
Covers Python 2.6 and 3.x
Learning Python
O'REILLY
Mark Lutz