CSC424 - ADVANCED DATABASE MANAGEMENT SYSTEMS

Semester: Spring 2019

Instructor: Suresh Melvin Sigera

Email: suresh.sigera@csi.cuny.edu

AGENDA

- Flow/Structure of the course, and expectations
- Homework, Labs, Exams, and Grading
- Development Environment, and Dev Tools
- Week 1 Lecture
- QA session

SEMESTER OVERVIEW

- Week 1 – Course Overview/Introduction to DBMS, review on SQL,

  and introduction to Python
- Week 2 – Continue to learn Python
- Week 3 – Introducing Psycopg, and CRUD using Psycopg
- Week 4 – Quiz 1, Python JSON, ACID vs CAP theorem, Introduction NOSQL database, and MongoDB
- Week 5 – MongoDB Performance, Fault tolerance, and deployment
- Week 6 – Understanding the basics, and CRUD operations

SEMESTER OVERVIEW

- Week 7 – Midterm review
- Week 8 – Midterm
- Week 9 – How to structure documents (schemas, and relations)
- Week 10 – Introducing PyMongo, and CRUD using PyMongo
- Week 11 – MongoDB security
- Week 12 – Quiz 2, and MongoDB Transactions

SEMESTER OVERVIEW

- Week 13 – Spring recess. No classes.
- Week 14 – Amazon Web Services (AWS)
- Week 15 – Review
- Week 16 – Final Exam

GRADING

- Attendance : 5%
- Homework/Labs : 30%
- Quiz 1 : 10%
- Midterm: 15%
- Quiz 2 : 10%
- Final: 30%

DEVELOPMENT ENVIRONMENT, AND DEV TOOLS

Throughout this course, I will be using **PyCharm** in an **Ubuntu** or **Windows** environment for all of the demos. **Make sure to install the following software on your personal computer. Do not put this off until your first assignment is due!**

Software	Download URL	Version
Anaconda Python	https://www.anaconda.com/download/	Python 3.7 version
PostgreSQL	https://www.enterprisedb.com/thank-you-downloading-postgresql?anid=1256152	11.1
PyCharm	https://www.jetbrains.com/pycharm-edu/	
MongoDB	https://www.mongodb.com/download-center Click on the server tab	4.0.5
Robo3t	https://download.robomongo.org/1.2.1/windows/robo3t-1.2.1-windows-x86_64-3e50a65.exe	1.2

COURSE OUTCOMES

- Upon successful completion of the course, the student will be able to:
  - Differentiate database systems from file systems by enumerating the features provided by database systems and describe each in both function and benefit.
  - Analyze an information storage problem and derive an information model expressed in the form of an entity relation diagram and other optional analysis forms, such as a data dictionary.
  - Demonstrate an understanding of the NOSQL data model.
  - Use a desktop database package to create, populate, maintain, and query a database.
  - Demonstrate a rudimentary understanding of programmatic interfaces to a database and be able to use the basic functions of one such interface.
  - Use Python packages such as PYMongo to interact with NOSQL Databases

LECTURE 1 – WEEK 1

DBMS & PostgreSQL Overview

DATA VS. INFORMATION

Data	Information

• Raw facts
  • Raw data - Not yet been processed to reveal the meaning
• Building blocks of information
• Data management
  • Generation, storage, and retrieval of data
	• Produced by processing data
	• Reveals the meaning of data
	• Enables knowledge creation
	• Should be accurate, relevant, and timely to enable good decision making

DATABASE

- Shared, integrated computer structure that stores a collection of
  - End-user data - Raw facts of interest to end user
  - **Metadata** : Data about data, which the end-user data are integrated and managed
    - Describe data characteristics and relationships
- Database management system (**DBMS**)
  - Collection of programs
  - Manages the database structure
  - Controls access to data stored in the database

ROLE OF THE DBMS

- Intermediary between the user and the database
- Enables data to be shared
- Presents the end user with an integrated view of the data
- Receives and translates application requests into
- operations required to fulfill the requests
- Hides database's internal complexity from the application programs and users

ADVANTAGES OF THE DBMS

- Better data integration and less data inconsistency
  - Data inconsistency: Different versions of the same data appear in different places
- Increased end-user productivity
- Improved
  - Data sharing
  - Data security
  - Data access
  - Decision making
    - **Data quality**: Promoting accuracy, validity, and timeliness of data

TYPES OF DATABASES

- **Single-user** database: Supports one user at a time
  - Desktop database: Runs on PC
- **Multiuser** database: Supports multiple users at the same time
  - Workgroup databases: Supports a small number of users or a specific department
  - **Enterprise database**: Supports many users across many departments
- **Centralized** database: Data is located at a single site
- **Distributed** database: Data is distributed across different sites
- **Cloud** database: Created and maintained using cloud data services that provide defined performance measures for the database

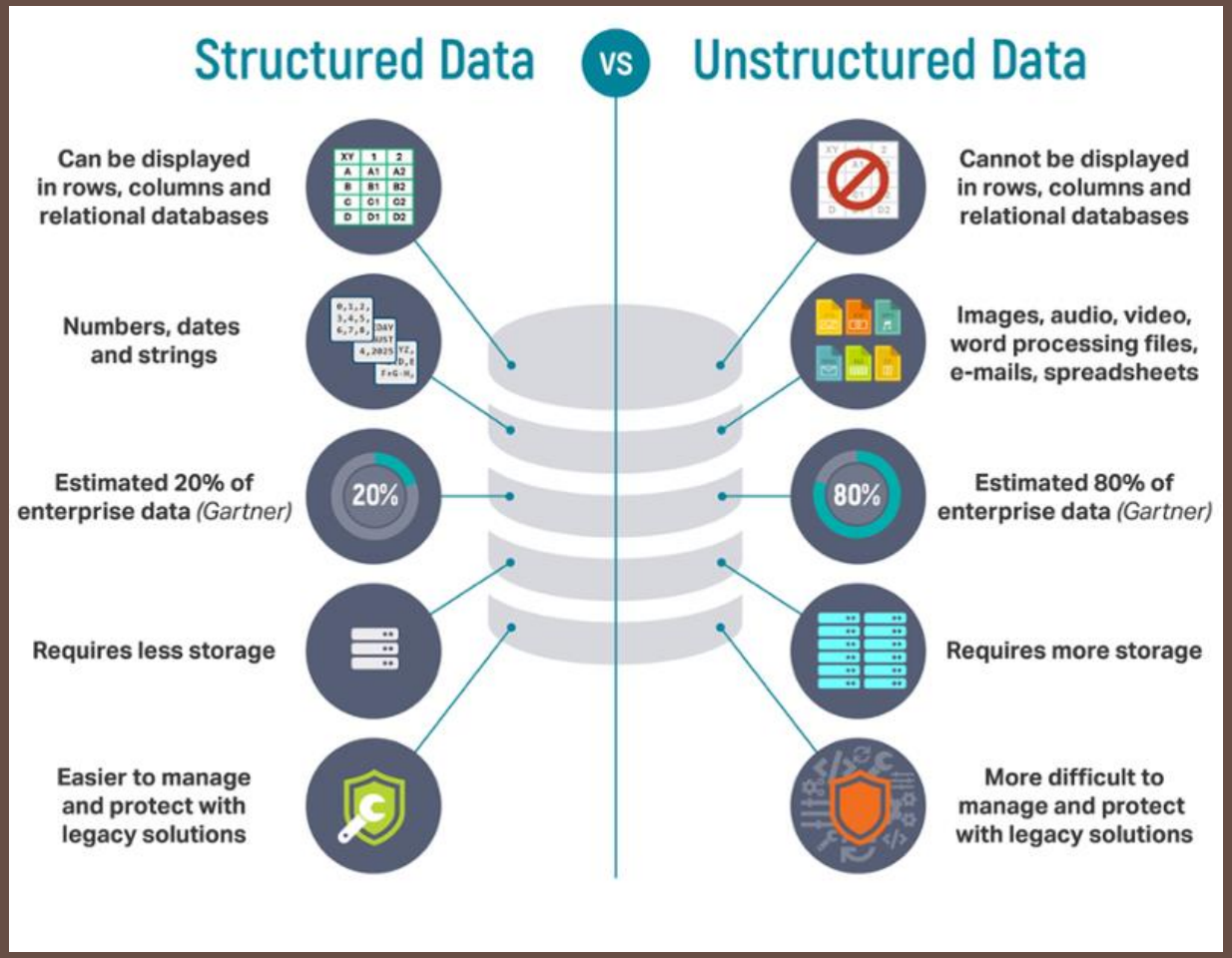
TYPES OF DATABASES

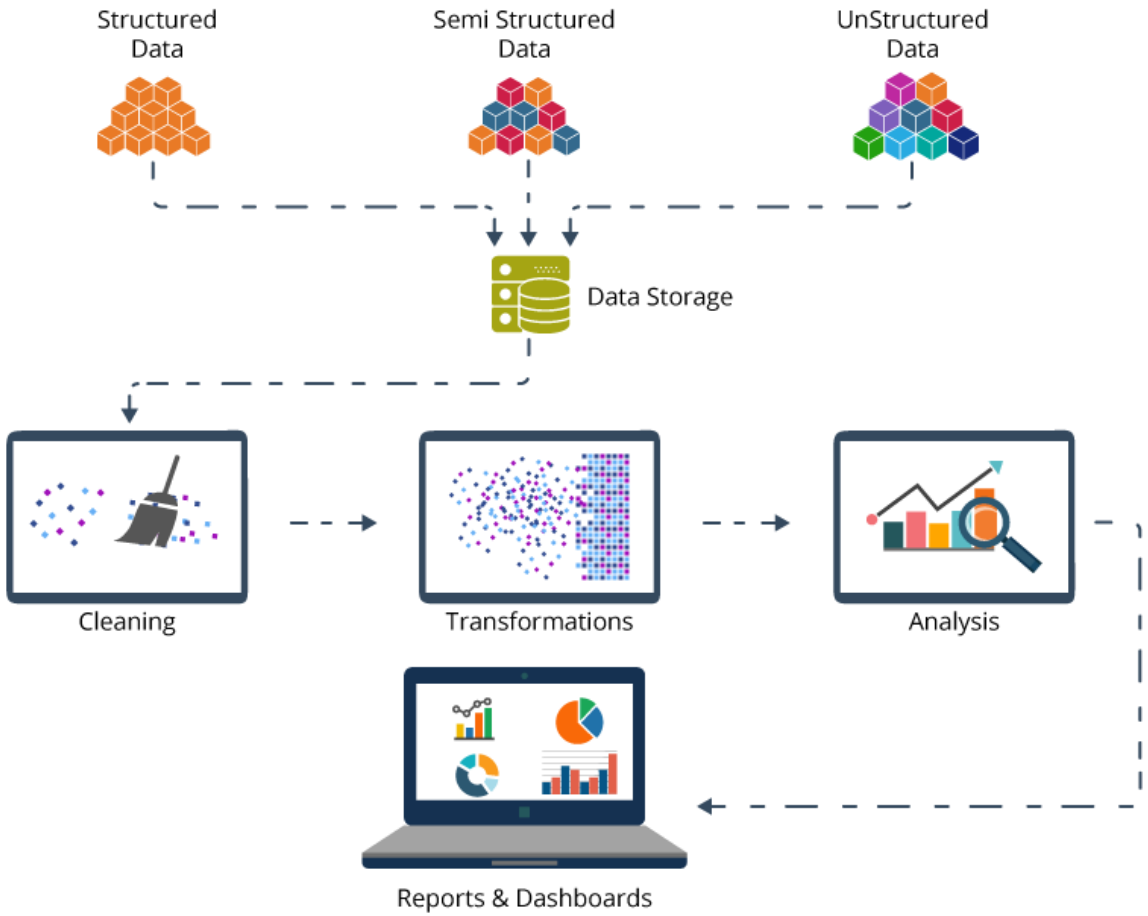
- General-purpose databases: Contains a wide variety of data used in multiple disciplines
- Discipline-specific databases: Contains data focused on specific subject areas
- Operational database: Designed to support a company's day-to-day operations
- Analytical database: Stores historical data and business metrics used exclusively for tactical or strategic decision making
  - Data warehouse: Stores data in a format optimized for decision support
- Online analytical processing (**OLAP**)
  - Enable retrieving, processing, and modeling data from the data warehouse
- Business intelligence: Captures and processes business data to generate information that support decision making

TYPES OF DATABASES

- **Unstructured** data: It exists in their original state
- **Structured** data: It results from formatting
- **Semistructured** data: Processed to some extent
  - Structure is applied based on type of processing to be performed
- **Extensible** Markup Language (**XML**)
  - Represents data elements in textual format

TYPES OF DATABASES





DATABASE DESIGN

- Focuses on the design of the database structure that will be used to store and manage end-user data
- Well-designed database
  - Facilitates data management
  - Generates accurate and valuable information
- Poorly designed database causes difficult-to-trace errors

DATA REDUNDANCY

- Unnecessarily storing same data at different places
  - Physically distributed among multiple storage facilities
- **Islands of information** :
  - Scattered data locations Increases the probability of having different versions of the same data

DATA REDUNDANCY IMPLICATIONS

- Poor data security
- Data inconsistency
- Increased likelihood of data-entry errors when complex entries are made in different files
- **Data anomaly** : Develops when not all of the required changes in the redundant data are made successfully

TYPES OF DATA ANOMALY

- **Update Anomalies**
- **Insertion Anomalies**
- **Deletion Anomalies**

LACK OF DESIGN AND DATA-MODELING SKILLS

- Evident despite the availability of multiple personal productivity tools being available
- Data-modeling skills is vital in the data design process
- Good data modeling facilitates communication between the designer, user, and the developer
  - **CAN YOU THINK OF MVC AND MVT?**

WORKING WITH DATABASES

- Most of the time, you will work with existing databases
- However, you still need to know the schema
- We will concentrate on four tasks
  - Extracting data
  - Changing data
  - Inserting data
  - Deleting data

WHY POSTGRESQL?

- PostgreSQL is free, and it’s used by NASA, US Navy, Netflix, WeChat, Facebook etc…
- PostgreSQL is the most advanced open source relational database.
- PostgreSQL isn't just relational, it's object-relational. This gives it some advantages over other open source SQL databases like MySQL, MariaDB and Firebird
- There's an extensive list of data types that PostgreSQL supports. Besides the numeric, floating-point, string, boolean and date types you'd expect (and many options within these), PostgreSQL boasts uuid, monetary, enumerated, geometric, binary, network address, bit string, text search, XML, JSON, array, composite and range types, as well as some internal types for object identification and log location.
- It is well documented, https://www.postgresql.org/docs/11/index.html
- We will using the version `11.1` for all our examples

NOTABLE FEATURES

- Just In Time (**JIT**) Compilation of SQL Statements - This is a cutting edge feature in PostgreSQL: SQL statements can get compiled into native code for execution. It's well know how much Google V8 JIT revolutionized the JavaScript language. JIT in PostgreSQL 11 supports accelerating two important factors—expression evaluation and tuple deforming during query execution—and helps CPU bound queries perform faster
- Multi-version concurrency control (**MVCC**)
- Asynchronous replication - Products copy the data to the replica after the data is already written to the primary storage. Although the replication process may occur in near-real-time, it is more common for replication to occur on a scheduled basis. For instance, write operations may be transmitted to the replica in batches on a periodic basis (for example, every one minute). In case of a fail-over event, some data loss may occur

NOTABLE FEATURES

- Add extensions to convert JSONB data to/from PL/Perl and PL/Python - Python as a programming language continues to gain popularity. It is always among the top 5 in the TIOBE Index. One of the greatest features of PostgreSQL is that you can write stored procedures and functions in most of the popular programming languages, including Python (with PL/Python). Now it is also possible to transform **JSONB** (Binary JSON) data type to/from PL/Python. This feature was later made available for PL/Perl too. It can be a great add-on for organizations using PostgreSQL as a document store.
- Command line improvements in psql: autocomplete and quit/exit - **psql** has always been friendly to first time PostgreSQL users through the various options like autocomplete and shortcuts. There's an exception though: users may find it difficult to understand how to effectively quit from psql, and often attempt to use non-existing quit and exit commands. Eventually, they find \q or ctrl + D, but not without frustrating themselves first. Fortunately, that shouldn't happen anymore: among many recent fixes and improvements to psql is the addition of the intuitive quit and exit commands to safely leave this popular command line client.

RELATIONAL THEORY

- Data are stored in groups called '**tables**' or '**relations**'
- Tables consist of '**rows**' or '**records**' and '**columns**' or '**fields**'
- Rows are usually identified by a unique '**key**' which may be a single column or a group of columns
- **Columns** can be linked to other **tables** with similar columns
- Good design of the database structure or '**schema**' is critical, MVC
- "All the data in a row should be dependent on the **key**, the whole **key**, and nothing but the **key**."

SELECT - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`SELECT * FROM table_name;`	`SELECT * FROM actor;`	If you want to query data from all column, you can use an asterisk (*) as the shorthand for all columns.
`SELECT column1, column2, columnN FROM table_name;`	`SELECT first_name, last_name, last_update FROM actor;`	You specify a list of columns in the table from which you want to query data in the SELECT clause. You use a comma between each column in case you want to query data from multiple columns.

SELECT DISTINCT - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`SELECT DISTINCT column1 FROM table_name;`	`SELECT DISTINCT first_name FROM actor;`	The `DISTINCT` clause is used in the `SELECT` statement to remove duplicate rows from a result set. The `DISTINCT` clause keeps one row for each group of duplicates. You can use the `DISTINCT` clause on one or more columns of a table.
`SELECT DISTINCT column1, column2, columnN FROM table_name;`	`SELECT DISTINCT release_year FROM film;`	If you specify multiple columns, the `DISTINCT` clause will evaluate the duplicate based on the combination of values of those columns.

WHERE - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`SELECT column1, column2, columnN FROM table_name WHERE conditions;`	`SELECT last_name, first_name FROM customer WHERE first_name = 'Jamie';`	The **WHERE** clause appears right after the **FROM** clause of the **SELECT** statement. The conditions are used to filter the rows returned from the **SELECT** statement. PostgreSQL provides you with various standard operators to construct the conditions.

Operator	Description
=	Equal
>	Gather than
<	Less than
>=	Gather than or equal
<=	Less than or equal
<> OR !=	Not equal
AND	Logical AND
OR	Logical OR

WHERE - SYNTAX VS COMMAND

COMMAND	Explanation

`SELECT customer_id, payment_date`
`FROM payment`
`WHERE amount <= 1 OR amount >= 8;`
	If you want to know who paid the rental with amount is either less than 1 USD or greater than 8 USD, you can use the following query with OR operator.

LIMIT - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`SELECT * FROM table_name LIMIT n;`	`SELECT * FROM customer LIMIT 10;`	PostgreSQL **LIMIT** is used in the **SELECT** statement to get a subset of rows returned by the query. PostgreSQL returns n number of rows generated by the query. If n is zero or **NULL**, it produces the result that is same as omitting the **LIMIT** clause.
`SELECT * FROM table_name LIMIT n OFFSET m;`	`SELECT * FROM customer LIMIT 100 OFFSET 2;`	PostgreSQL first skips m rows before returning n rows generated by the query. If m is zero, PostgreSQL will behave like without the **OFFSET** clause. Because the order of the rows in the database table is unknown and unpredictable, when you use the **LIMIT** clause, you should always use the **ORDER BY** clause to control the order of rows. If you don't do so, you will get an unpredictable result.

ORDER BY - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation

`SELECT column1, column2`
`FROM table_name`
`ORDER BY column1 ASC,`
`column_2 DESC;`
	`SELECT first_name, last_name`
	`FROM customer`
	`ORDER BY first_name ASC;`
		When you query data from a table, PostgreSQL returns the rows in the order that they were inserted into the table. In order to sort the result set, you use the **ORDER BY** clause in the **SELECT** statement.
		
		The **ORDER BY** clause allows you to sort the rows returned from the **SELECT** statement in ascending or descending order based on criteria specified by different criteria.
		
		The following query sorts customers by the first name in ascending order.

BETWEEN - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`value BETWEEN low AND high;`	`value BETWEEN 7 AND 10;`	We use the **BETWEEN** operator to match a value against a range of values. The following illustrates the syntax of the **BETWEEN** operator. If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, or vice versa.

`SELECT column1, column1, columnN`
`FROM table_name`
`WHERE column1`
`BETWEEN value AND value;`
	`SELECT customer_id, payment_id, amount`
	`FROM payment`
	`WHERE amount`
	`BETWEEN 7 AND 10;`
		The following query selects any payment whose amount is between 7 and 10.

IN OPERATOR - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`value IN ( value1, value2 ...);`		You use the **IN** operator in the **WHERE** clause to check if a value matches any value in a list of values. The syntax of the **IN** operator is as follows.
`SELECT column1, column1, columnN FROM table_name WHERE column1 IN (value1, value2) ORDER BY column1 DESC;`	`SELECT customer_id, rental_id, return_date FROM rental WHERE customer_id IN (1,2) ORDER BY return_date DESC;`	The expression returns true if the value matches any value in the list i.e., value1, value2, etc. The list of values is not limited to a list of numbers or strings but also a result set of a **SELECT** statement as shown in the following query. Suppose you want to know the rental information of customer id 1 and 2, you can use the **IN** operator in the **WHERE** clause as follows.

NOT IN OPERATOR - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation
`value NOT IN ( value1, value2 ...);`		You can combine the IN operator with the **NOT** operator to select rows whose values do not match the values in the list.
`SELECT column1, column1, columnN FROM table_name WHERE column1 NOT IN (value1, value2);`	`SELECT customer_id, rental_id, return_date FROM rental WHERE customer_id NOT IN (1,2)`	The following statement selects rentals of customers whose customer id is not 1 or 2.

LIKE - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation

`SELECT column1, column2, columnN`
`FROM table_name`
`WHERE column1`
`LIKE 'string%';`
	`SELECT first_name, last_name`
	`FROM customer`
	`WHERE first_name`
	`LIKE 'Jen%';`
		Suppose the store manager asks you find a customer that he does not remember the name exactly. He just remembers that customer's first name begins with something like Jen. How do you find the exact customer that the store manager is asking? You may find the customer in the customer table by looking at the first name column to see if there is any value that begins with Jen. It is kind of tedious because there many rows in the customer table.
		
		Notice that the **WHERE** clause contains a special expression: the first_name, the **LIKE** operator and a string that contains a percent (%) character, which is referred as a pattern.

GROUP BY - SYNTAX VS COMMAND

SYNTAX	COMMAND	Explanation

`SELECT column1,`
`AGGERATE_FUNCTION(column2)`
`FROM table_name`
`GROUP BY column1;`
	`SELECT customer_id,`
	`SUM(amount)`
	`FROM payment`
	`GROUP BY customer_id;`
		The **GROUP BY** clause divides the rows returned from the **SELECT** statement into groups. For each group, you can apply an aggregate function e.g., to calculate the sum of items or count the number of items in the groups.
		
		The **GROUP BY** clause is useful when it is used in conjunction with an aggregate function. For example, to get how much a customer has been paid, you use the **GROUP BY** clause to divide the payments table into groups; for each group, you calculate the total amounts of money by using the **SUM** function as the following query.

QA

ABOUT PYTHON

- Development started in the 1980's by Guido van Rossum.
  - Only became popular in the last decade or so.
- Python 2.x currently dominates, but Python 3.x is the future of Python.
- Interpreted, very-high-level programming language.
- Supports a multitude of programming paradigms.
  - OOP, functional, procedural, logic, structured, etc.
- General purpose.
  - Very comprehensive standard library includes numeric modules, crypto services, OS interfaces, networking modules, GUI support, development tools, etc.

NOTABLE FEATURES

- Easy to learn
- Supports quick development
- Cross-platform
- Open Source
- Extensible
- Embeddable
- Large standard library and active community
- Useful for a wide variety of applications

INTERPRETER

- The standard implementation of Python is interpreted.
  - You can find info on various implementations here.
- The interpreter translates Python code into bytecode, and this bytecode is executed by the Python VM (similar to Java).
- Two modes: normal and interactive.
  - Normal mode: entire .py files are provided to the interpreter.
  - Interactive mode: read-eval-print loop (REPL) executes statements piecewise.

INTERPRETER: NORMAL MODE

Let's write our first Python program!

In our favorite editor, let's create helloworld.py with the following contents:

```
print "Hello, World!"
```

From the terminal:

```
$ python helloworld.py
Hello, World!
```

Note: In Python 2.x, print is a statement. In Python 3.x, it is a function. If you want to get into the 3.x habit, include at the beginning:

```
from __future__ import print_function
```

Now, you can write

```
print("Hello, World!")
```

INTERPRETER: NORMAL MODE

Let's include a she-bang in the beginning of helloworld.py:

```
#!/usr/bin/env python
print "Hello, World!"
```

Now, from the terminal:

```
$ ./helloworld.py
Hello, World!
```

PYTHON TYPING

- Python is a strongly, dynamically typed language.
- Strong Typing
  - Obviously, Python isn't performing static type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
  - Example: 2 + "four" ← not going to fly
- Dynamic Typing
  - All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.

Let's start by looking at Python's built-in data types.

SOME LANGUAGES (JAVA, C++, SWIFT) ARE STATICALLY TYPED

```
String myVar = "Hello";


/* myVar has been declared as a String,
and cannot be assigned the integer
value 10 later.*/
myVar = 10; // Does not work!


/*"abc" is a String - so compatible
type and assignment works.*/
myVar = "abc"; // This is OK!
```

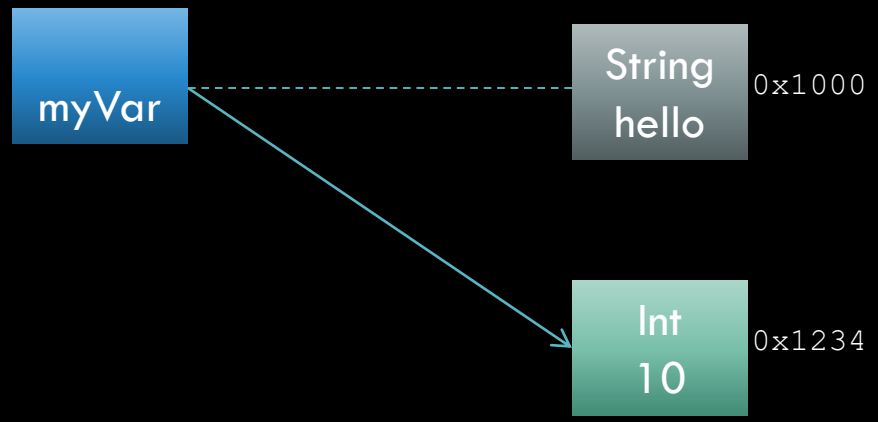
PYTHON, IN CONTRAST, IS DYNAMICALLY TYPED.

```
# The variable my_var is purely a
reference to a string object with value
hello. No type is "attached" to my_var.

my_var = 'hello';

# The variable my_var is now pointing
to an integer object with value 10.

my_var = 10;
```



PYTHON, IN CONTRAST, IS DYNAMICALLY TYPED.

We can use the built-in **type()** function to determine the type of the object currently referenced by a variable.

Remember: variables in Python do not have an inherent static type. Instead, when we call **type(my_var)** Python looks up the object my_var is **referencing** (pointing to), and returns the **type of the object** at that memory location.

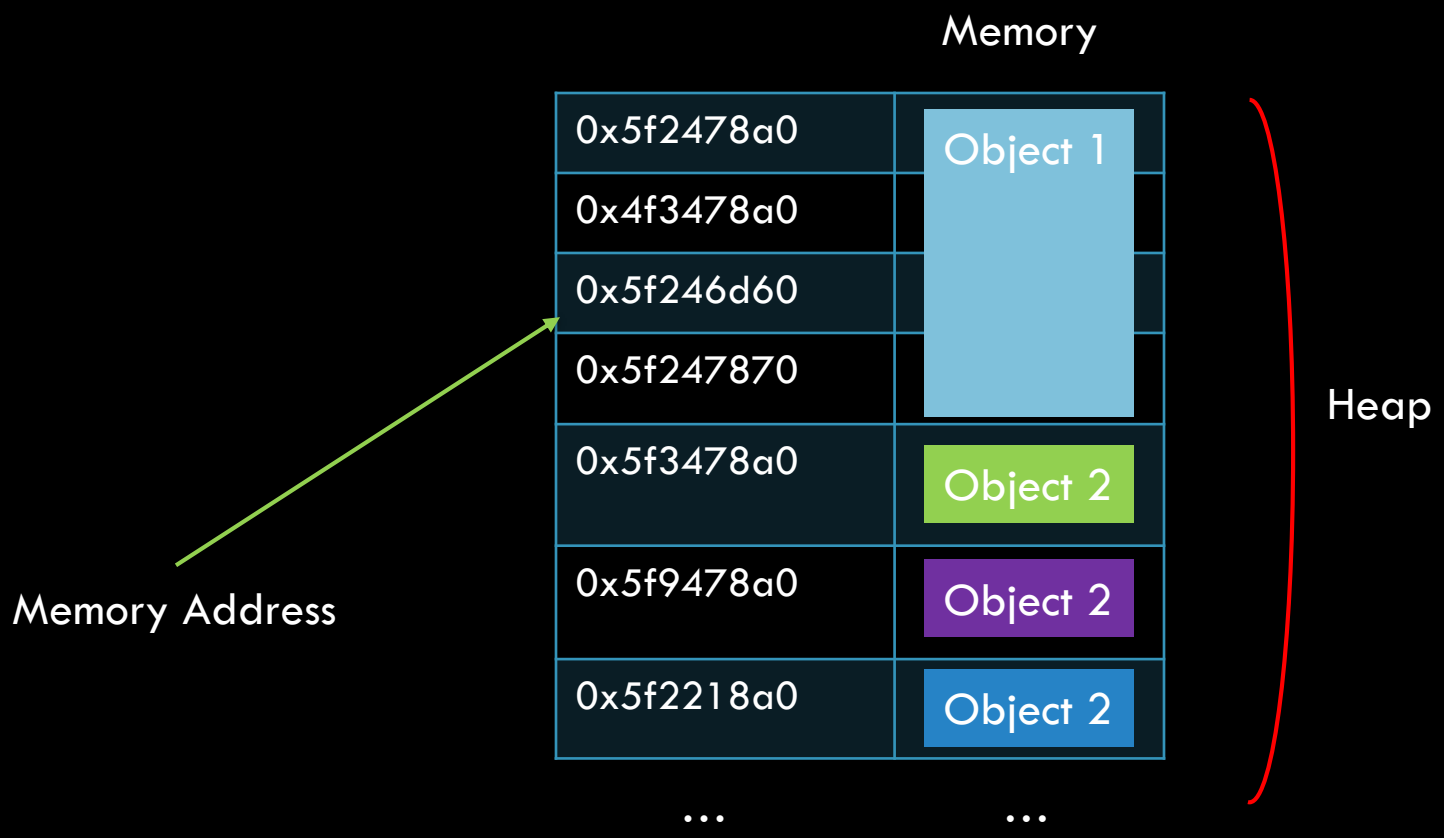
In Python, we can find out the memory address referenced by a variable by using the id() function. Example a = 10 print(hex(id(a))) This will return a base-10 number. We can convert this base-10 number to hexadecimal, by using the hex() function.

```
print(hex(id(my_var)))
```

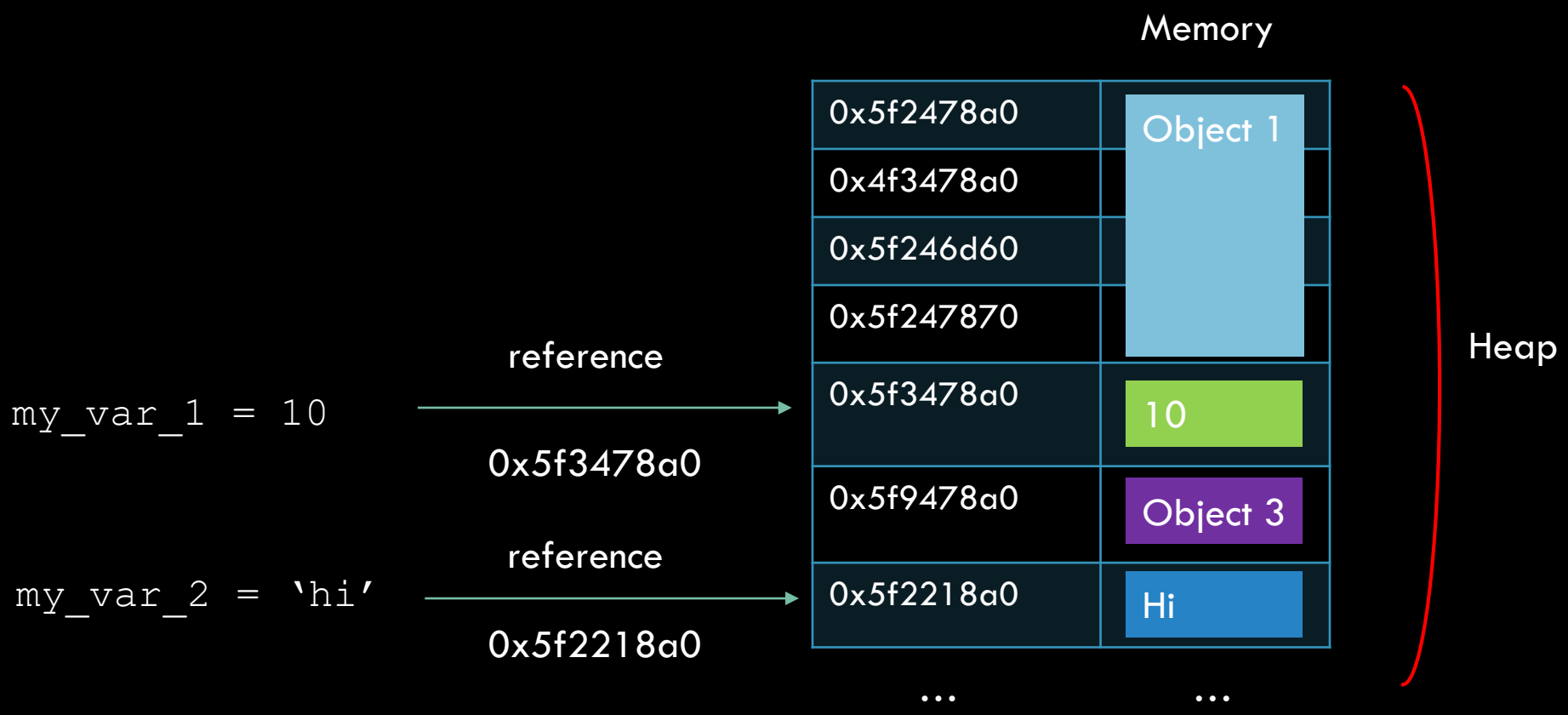
PYTHON, IN CONTRAST, IS DYNAMICALLY TYPED.

```
>>>x = "hello"
>>>type(x)
<class 'str'>
>>>hex(id(x))
'0x1d6e691c420'
>>>x = 10
>>>type(x)
<class 'int'>
>>>hex(id(x))
'0x5f246d60'
```

VARIABLES IN PYTHON

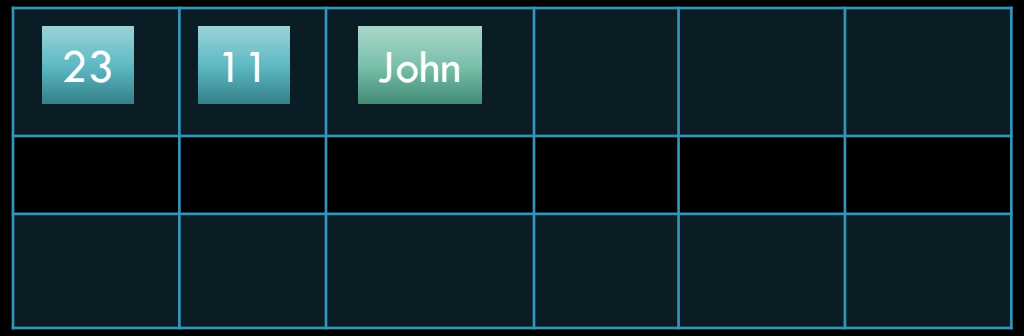


VARIABLES IN PYTHON



Read as my_var_1 *references to the object* at 0x5f3478a0

IMMUTABLE OBJECTS, AND GARBAGE COLLECTION



a = 23
b = 11
name = “John“
b = 15

Because numbers are immutable, "b" changes location to the new value. When there is no reference to a memory location the value fades away, and the location is free to use again. This process known as **garbage collection**.

CREATING VARIABLES IN PYTHON

TYPE	PYTHON CODE
bool	`is_cat = True` `this_cat = False`
int	`age= 10`
float	`gravity = 9.8`
tuple	`score = 1,2,3` `score = (1,2,3)`
str	`name = "Stacy"`

TYPE	PYTHON CODE
list	`[1,3,5, 'John',1.1]`
set	`{1,4,3}`

dictionaries	`{'Stacy': 100,`
	`'John': 90,`
	`'Alex': 45,`
	`'Mike': 78`
	`}`

DATATYPES, AND MUTABILITY

Type	Mutable/Immutable
bool	Immutable
int	Immutable
float	Immutable
tuple	Immutable
str	Immutable
frozen sets	Immutable
user-defined classes	Immutable

Type	Mutable/Immutable
list	immutable
sets	immutable
dictionaries	immutable
user-defined classes	immutable

Note:
Mutable = A mutable object can be changed after it's created
Immutable = Object cannot be change after it's created

NONE = NULL

Python has a special null value called None, spelled with a capital N. None is frequently used to represent the absence of a value. The Python REPL never prints None results, so typing None into the REPL has no effect:

The null value None can be bound to variable names just like any other object.

```
a = None
```

NUMERIC TYPES

The subtypes are int, long, float and complex.

- Their respective constructors are int(), long(), float(), and complex().

- All numeric types, except complex, support the typical numeric operations you'd expect to find (a list is available here).
- Mixed arithmetic is supported, with the "narrower" type widened to that of the other. The same rule is used for mixed comparisons.

NUMERIC TYPES

- Numeric
  - **int**: equivalent to C's long int in 2.x but unlimited in 3.x.
  - **float**: equivalent to C's doubles.
  - **long**: unlimited in 2.x and unavailable in 3.x.
  - **complex**: complex numbers.

- Supported operations include constructors (i.e. int(3)), arithmetic, negation, modulus, absolute value, exponentiation, etc.

```
$ python
>>> 3 + 2
5
>>> 18 % 5
3
>>> abs(-7)
7
>>> float(9)
9.0
>>> int(5.3)
5
>>> complex(1,2)
(1+2j)
>>> 2 ** 8
256
```

STRINGS

Lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there. Strings consist only of characters and are immutable (cannot change individual elements), while lists are flexible container objects that hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy, too, as we see in the following examples.

Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will. Lists can contain different types of objects and are more flexible than an array of C structs or Python arrays (available through the external array module) because arrays are restricted to containing objects of a single type.

STRINGS

SYNTAX	Explanation

`>>>'Snap'`
`'Snap'`
`>>>"Crackle"`
`'Crackle'`
	You make a Python string by enclosing characters in either single quotes or double quotes, as demonstrated in the following

`>>>"'Nay,' said the naysayer."`
`"'Nay,' said the naysayer."`
`>>> 'The rare double quote in captivity: "' 'The rare double quote in captivity: "'`
`>>> 'A "two by four" is actually 1 1/2" x 3 1/2".' 'A "two by four is" actually 1 1/2" x 3 1/2".'`
`>>> "'There's the man that shot my paw!' cried the limping hound."`
`"'There's the man that shot my paw!' cried the limping hound."`
	Why have two kinds of quote characters? The main purpose is so that you can create strings containing quote characters. You can have single quotes inside double-quoted strings, or double quotes inside single-quoted strings

`>>>'''Boom!'''`
`'Boom'`
`>>>"""Eek!"""`
`'Eek!'`
	You can also use three single quotes (''') or three double quotes (""")

SYNTAX	Explanation
`>>>poem = '''There was a Young Lady of Norway, ... Who casually sat in a doorway; ... When the door squeezed her flat, ... She exclaimed, "What of that?" ... This courageous Young Lady of Norway.'''` `>>>`	Triple quotes aren't very useful for short strings like these. Their most common use is to create multiline strings, like this classic poem from Edward Lear

`>>>str(98.6)`
`'98.6'`
`>>> str(1.0e4)`
`'10000.0'`
`>>> str(True)`
`'True'`
	You can convert other Python data types to strings by using the str() function

`>>>'Release the kraken! ' + 'At once!'` `'Release the kraken! At once!'`	You can combine literal strings or string variables in Python by using the + operator, as demonstrated here
`>>>'Release the kraken! ' + 'At once! '` `'Release the kraken! At once!'`	ou can combine literal strings or string variables in Python by using the + operator

LIST – A SEQUENCE OF OBJECTS

Lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there. Strings consist only of characters and are immutable (cannot change individual elements), while lists are flexible container objects that hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy, too, as we see in the following examples.

Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will. Lists can contain different types of objects and are more flexible than an array of C structs or Python arrays (available through the external array module) because arrays are restricted to containing objects of a single type.

LIST – A SEQUENCE OF OBJECTS

SYNTAX	Explanation
`>>>[1, 9, 8]` `[1, 9, 8]`	Here is a list of three numbers
`>>>a = ["apple", "orange", "pear"]`	And here is a list of three strings
`>>>a[1]` `"orange"`	We can retrieve elements by using square brackets with a zero-based index
`>>>a[1] = 7` `>>>a ['apple', 7, 'pear']`	We can replace elements by assigning to a specific element See how lists can be heterogeneous with respect to the type of the contained objects. We now have a list containing an str, an int, and another str

SYNTAX	Explanation
`>>>b = []`	It's often useful to create an empty list, which we can do using empty square brackets

`>>>b.append(1.618)`
`>>>b [1.618]`
`>>>b.append(1.414)`
`[1.618, 1.414]`
	We can modify the list in other ways. Let's add some floats to the end of the list using the append() method

`>>>list("characters")` `['c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', 's']`	There is also a list constructor, which can be used to create lists from other collections, such as strings

`>>>names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']`
`>>> del names[2]`
`>>> names`
`['Alice', 'Beth', 'Dee-Dee', 'Earl']`
	Deleting elements from a list is easy, too. You can simply use the del statement

TUPLES

SYNTAX	Explanation
`>>>1, 2, 3` `(1, 2, 3)`	The tuple syntax is simple—if you separate some values with commas, you automatically have a tuple
`>>>(1, 2, 3)` `(1, 2, 3)`	As you can see, tuples may also be (and often are) enclosed in parentheses
`>>>()` `()`	The empty tuple is written as two parentheses containing nothing

`>>>42,`
`(42,)`
`>>> (42,)`
`(42,)`
	So, you may wonder how to write a tuple containing a single value. This is a bit peculiar—you have to include a comma, even though there is only one value.

SYNTAX	Explanation
`>>(42)` `42`	Simply adding parentheses won't help: (42) is exactly the same as 42. One lonely comma, however, can change the value of an expression completely

`>>>tuple([1, 2, 3])`
`(1, 2, 3)`
`>>> tuple('abc')`
`('a', 'b', 'c')`
`>>> tuple((1, 2, 3))`
`(1, 2, 3)`
	The tuple function works in pretty much the same way as list: it takes one sequence argument and converts it to a tuple

`>>>score = (20,10,11)`
`(20, 10, 11)`
`>>del score[1]`
`Traceback (most recent call last):`
`  File "<stdin>", line 1, in <module>`
`TypeError: 'tuple' object doesn't support item deletion`
`>>del score`
	Removing individual tuple elements is not possible. To explicitly remove an entire tuple, just use the del statement to reduce an object's reference count.

SLICING

SYNTAX	Explanation
`>>>a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']` `(1, 2, 3)`	The tuple syntax is simple—if you separate some values with commas, you automatically have a tuple
`>>> (1, 2, 3)` `(1, 2, 3)`	As you can see, tuples may also be (and often are) enclosed in parentheses
`>>> ()` `()`	The empty tuple is written as two parentheses containing nothing

`>>> 42,`
`(42,)`
`>>> (42,)`
`(42,)`
	So, you may wonder how to write a tuple containing a single value. This is a bit peculiar—you have to include a comma, even though there is only one value.

SYNTAX	Explanation
`>> (42)` `42`	Simply adding parentheses won't help: (42) is exactly the same as 42. One lonely comma, however, can change the value of an expression completely

`>>> tuple([1, 2, 3])`
`(1, 2, 3)`
`>>> tuple('abc')`
`('a', 'b', 'c')`
`>>> tuple((1, 2, 3))`
`(1, 2, 3)`
	The tuple function works in pretty much the same way as list: it takes one sequence argument and converts it to a tuple

`>>> score = (20,10,11)`
`(20, 10, 11)`
`>> del score`
`Traceback (most recent call last):`
`File "<ipython-input-27-d2d780e36333>", line 1, in <module>`
`score`
`NameError: name 'score' is not defined`
	Removing individual tuple elements is not possible. To explicitly remove an entire tuple, just use the del statement to reduce an object's reference count.

DICTIONARY

Python's name for associative arrays or maps, which it implements by using hash tables. Dictionaries are amazingly useful, even in simple programs.

Another extremely important data structure available in Python is called a dictionary. A dictionary is similar to a list in that it allows you to refer to a collection of data by a single variable name. However, it differs from a list in one fundamental way. In a list, order is important, and the order of the elements in a list never changes (unless you explicitly do so). Because the order of elements in a list is important, you refer to each element in a list using its index (its position within the list). In a dictionary, the data is represented in what are called key/value pairs . The syntax of a dictionary looks like this:

{<key>:<value>, <key>:<value>, ..., <key>:<value>}

DICTIONARY

SYNTAX	Explanation

>>>houseDict = {
'color' : 'blue',
'style' : 'colonial',
'numberOfBedrooms' : 4,
'garage' : True,
'burglarAlarm' : False,
'streetNumber' : 123,
'streetName' : 'Any Street',
'city' : 'Anytown',
'state' : 'CA',
'price' : 625000
}
	We are naming this dictionary houseDict to make it clear that this is a dictionary. we are using a name like this as an extension to our naming convention. In this example, all the keys of this dictionary are strings, which is a very common practice.

>>>houseDict >>>{'color': 'blue', 'style': 'colonial', 'numberOfBedrooms': 4, 'garage': True, 'burglarAlarm': False, 'streetNumber': 123, 'streetName': 'Any Street', 'city': 'Anytown', 'state': 'CA', 'price': 625000}	However, the keys in a dictionary can be of any type of immutable data—integers, floats, Booleans, and tuples can also be used as keys.
>>>print(houseDict['color']) Blue	We want to access any piece of data in a dictionary, we do it by using a key as an index

SYNTAX	Explanation
>>>houseDict['price'] = 575000 print(houseDict) {'color': 'blue', 'style': 'colonial', 'numberOfBedrooms': 4, 'garage': True, 'burglarAlarm': False, 'streetNumber': 123, 'streetName': 'Any Street', 'city': 'Anytown', 'state': 'CA', 'price': 575000} >>>	To assign a new value for an existing key in a dictionary, we use an assignment statement, like this.
>>>houseDict['numberOfBathrooms'] = 2.5 **print(houseDict)** **{'color': 'blue',** **'style': 'colonial',** **'numberOfBedrooms': 4,** **'garage': True,** **'burglarAlarm': False,** **'streetNumber': 123,** **'streetName': 'Any Street',** **'city': 'Anytown', 'state': 'CA',** **'price': 575000,** **'numberOfBathrooms': 2.5}**	o add a new key/value pair into a dictionary, we use an assignment statement the same way. If the key we are specifying does not exist in the dictionary, then the key/value pair is added to the dictionary.

QA