BERTELSMANN DATA SCIENCE CHALLENGE SCHOLARSHIP COURSE NOTES

by

KALHARI LIYANAGAMA

STATISTICS

INTRO TO STATISTICAL RESEARCH METHODS

CONSTRUCTS

Construct is anything that is difficult to measure because it can be defined and measured in many different ways.

Operational definition of a construct is the unit of measurement we are using for the construct. Once we operationally define something it is no longer a construct.

Volume is a construct. volume is the space something takes up but we haven't defined how we are measuring that space. (i.e. liters, gallons) Volume in liters is not a construct because it is operationally defined. Minutes is already operationally defined; there is no ambiguity in what we are measuring.

POPULATION & SAMPLE

- Population ➔ all the individuals in a group
- Sample ➔ some of the individuals in a group
- parameter ➔ a characteristic of the population
- statistic ➔ a characteristic of the sample
- μ ➔ mean of a population
- x bar ➔ mean of a sample

Difference between the sample & population averages is known as sampling error.

Larger sample size is better.

In a random sample each subject has an equal chance of being selected, and it's more likely to approximate the population.

OBSERVATIONAL STUDIES & CONTROLLED EXPERIMENTS

In an experiment, manner in which researchers handle subjects is called a treatment. Researchers are specifically interested in how different treatments might yield differing results.

An observational study is when an experimenter watches a group of subjects and does not introduce a treatment. A survey is an observational study.

Surveys are used a lot in social and behavioral science

Advantages	Disadvantages
Easy way to get infor on a population	Untruthful responses
Relatively inexpensive	Biased responses
Conducted remotely	Respondents not understanding the questions
Anyone can access & analyze survey results	Respondents refusing to answer

Correlation does not imply causation

- Show relationships ➔ Observational studies (Surveys)
- Show causation➔ Controlled experiment

Hypotheses are statements about the relationships between variables.

Independent Variable is the variable that experimenters choose to manipulate; it is usually plotted along the x-axis of a graph. AKA predictor variable.

Dependent Variable is the variable that experimenters choose to measure during an experiment; it is usually plotted along the y-axis of a graph. AKA outcome.

Extraneous factors or Lurking variables are things that can impact the outcome of data analysis. It's difficult to account for every extraneous factor. Can minimize the effect of certain variables on results of a experiment via random assignment.

In an experiment researcher manipulates independent variable, measures changes in the dependent variable and controls lurking variables.

Treatment Group is the group of a study that receives varying levels of the independent variable. These groups are used to measure the effect of a treatment

Control Group is the group of a study that receives no treatment. This group is used as a baseline when comparing treatment groups

Placebo is something given to subjects in the control group so they think they are getting the treatment, when in reality they are getting something that causes no effect. (i.e. sugar pill)

Blinding is a technique used to reduce bias. Double blinding ensures that both those administering treatments and those receiving treatments do not know who is receiving which treatment.

VISUALIZING DATA

Frequency of a data set is the number of times a certain outcome occurs.

Frequency Table is where outcomes and frequency is tabulated.

A proportion is the fraction of counts over the total sample. A proportion can be turned into a percentage by multiplying the proportion by 100.

- RF = Relative Frequency = Proportion
- RF = Frequency/total number
- 0 <= RF <= 1
- Sum (RF) = 1
- Percentage = RF *100
- 0 <= Percentage <= 100%

Organize the data based on the questions you want to answer.

Histogram is a graphical representation of the distribution of data.

Frequency is on the y-axis & variable is on the x-axis.

Intersection of the axes is origin. Its Cartesian coordinates are (0,0) if we go zero & up in both axes.

It depends on what questions you want to answer.

discrete intervals (bins) are decided upon to form widths for our boxes. Bin size (interval size) is the interval in which you're counting the frequency. Shape of the histogram changes with the bin size. Adjusting the bin size of a histogram will compact (or spread out) the distribution.

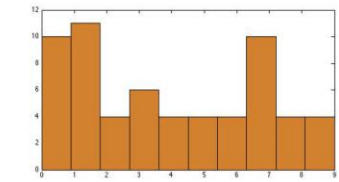


Figure 2.1: histogram of data set with bin size 1

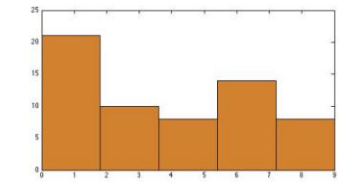


Figure 2.2: histogram of data set with bin size 2

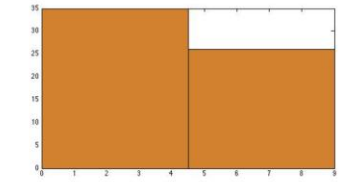
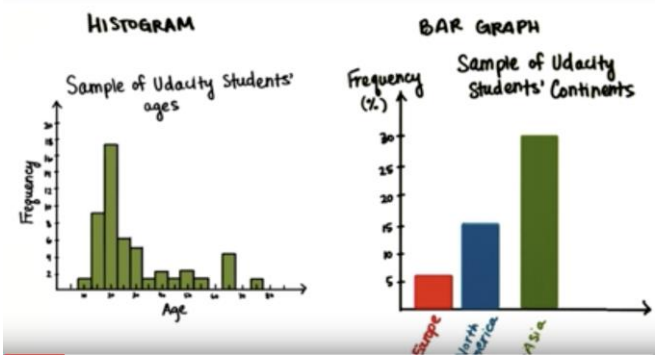


Figure 2.3: histogram of data set with bin size 5

Histogram is better for analyzing the shape of a distribution of data. Frequency table is better for calculating n.



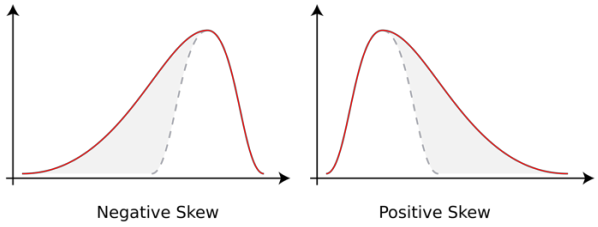
Histogram	Bar graph
Can choose any interval or bin size	spaces in between columns means that each of them is one distinct category
Can change the bin size	Not possible to change an interval like that
Order matters, it goes from low to high along x-axis	Order doesn't matter
Variable on x axis is numerical & quantitative	Variable on x axis is categorical or qualitative
Shape is very important	Shape is arbitrary

Biased Graphs

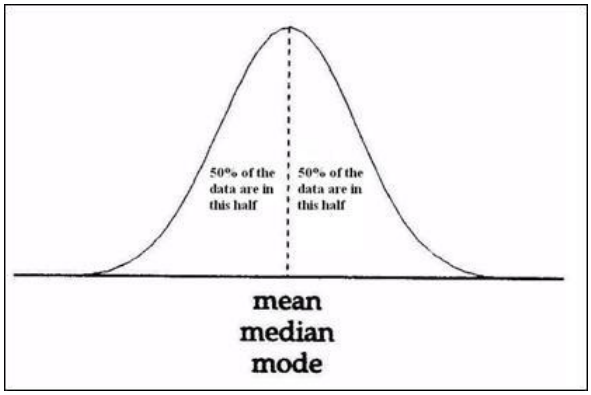
Skewed Distribution

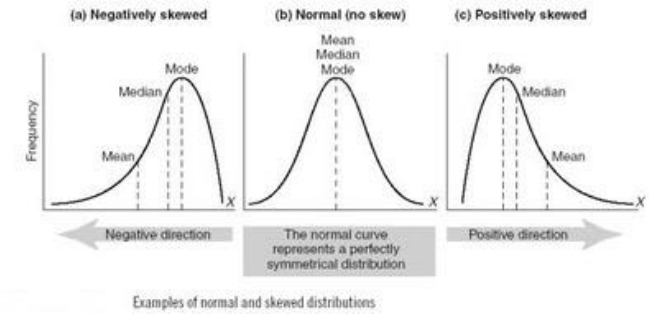
A positive skew is when outliers are present along the right most end of the distribution

A negative skew is when outliers are present along the left most end of the distribution

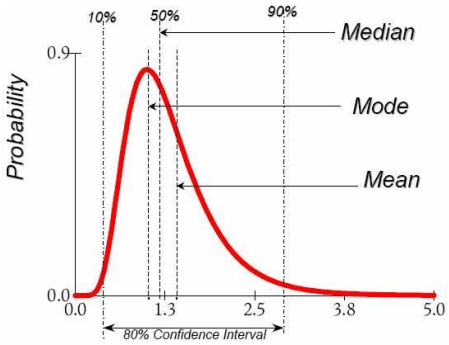


CENTRAL TENDENCY





Examples of normal and skewed distributions



Mean, median & mode are measures of center.

Median is the best measure of central tendency when you have highly skewed distributions.

In a normal distribution mean = median = mode.

- Σ (capital sigma) = total sum.
- f = frequency (count)
- p = proportion

MEAN

$$\bar{x} = \frac{\sum_{i=0}^{n} x_i}{n}$$

mean is the numerical average and can be computed by dividing the sum of all the data points by the number of data points.

mean is heavily affected by outliers; therefore, it is not a robust measurement. Outliers are values that are unexpectedly different from the other observed vales. They create skewed distributions by pulling the mean towards the outlier.

MEDIAN

median is the data point that is directly in the middle of the data set. If two numbers are in the middle, median is the average of the two. Median splits data in half. Data has to be in order for the median to be a useful value.

median is robust to outliers, therefore an outlier will not affect the value of the median

Median can be used to describe categorical data.

data set is odd ➔ n/2 = position in the data set the middle value is

data set is even ➔ $(x_k+x_{k+1})/n$ gives the median for the two middle data points

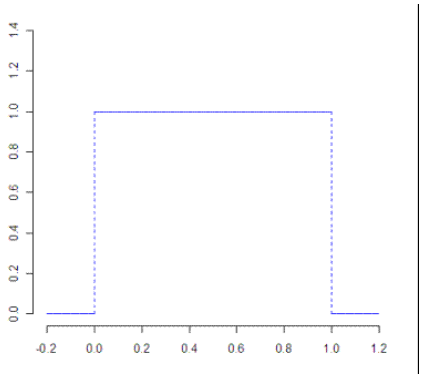
MODE

mode is the data point that occurs most frequently in the data set. (it could be a single number or a range)

mode is robust to outliers as well.

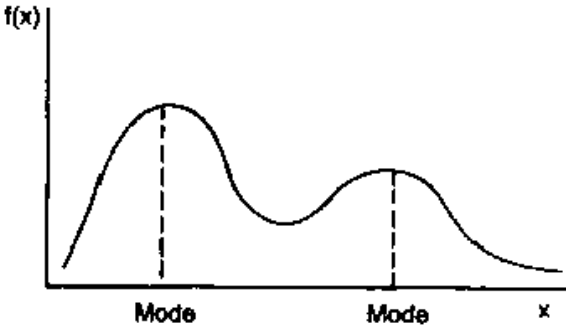
Mode can be used to describe any type of data, whether its numerical or categorical.

There is no mode in a uniform distribution



Although there is one value for which there is a maximum frequency, in terms of describing the 'modality' of a distribution, need to consider number of local peaks.

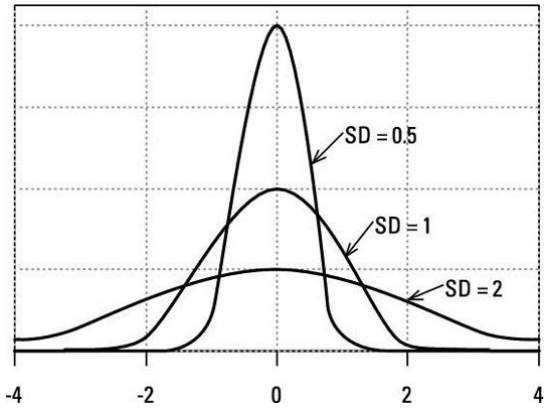
Bimodal distribution has two modes



Mode changes with the bin size. It depends on how you present data

	Mean	Median	Mode
Has a simple equation	Y		
Will always change if any data value changes	Y		
Not affected by change in bin size	Y	Y	
Not affected severely by outliers		Y	Y
Easy to find on a histogram			Y

VARIABILITY



All 3 distributions have same mean, median & mode. They differ in the way they spread out.

Range (max-min) to indicate how spread out the distribution is not robust, since it's based on only two data points and if they are outliers you get a wrong picture.

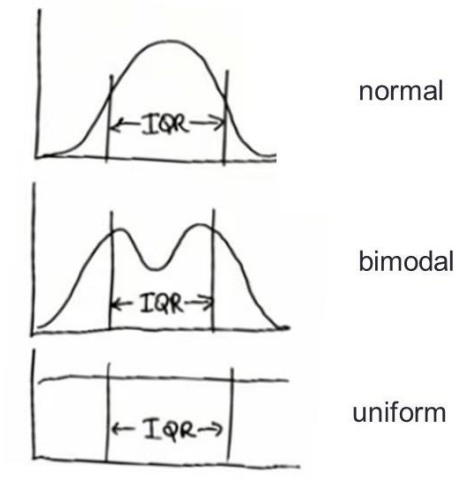
Interquartile range (IQR) is the distance between the 1st quartile and 3rd quartile and gives the range of the middle 50% of data. It's not affected by outliers.

IQR is found by: Q3 - Q1

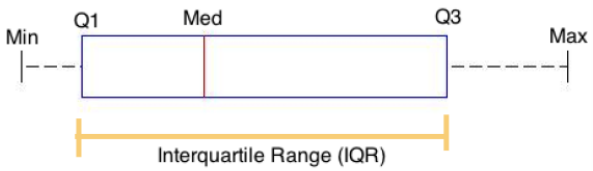
You can use the IQR to identify outliers:

- Upper outliers: Q3 + 1.5(IQR) Anything higher
- Lower outliers: Q1 - 1.5(IQR) Anything lower

Varying distributions can have the same IQR



A box plot (box & whisker plot) visualize quartiles & outliers. Outliers are represented as dots. It is a great way to show the 5 number summary of a data set in a visually appealing way. The 5 number summary consists of minimum, first quartile, median, third quartile, and maximum.



variance is the average of the squared differences from the mean. Standard deviation is the square root of the variance and is used to measure distance from the mean.

$$\sigma^2 = \frac{\sum_{i=0}^{n}(x_i - \bar{x})^2}{n}$$

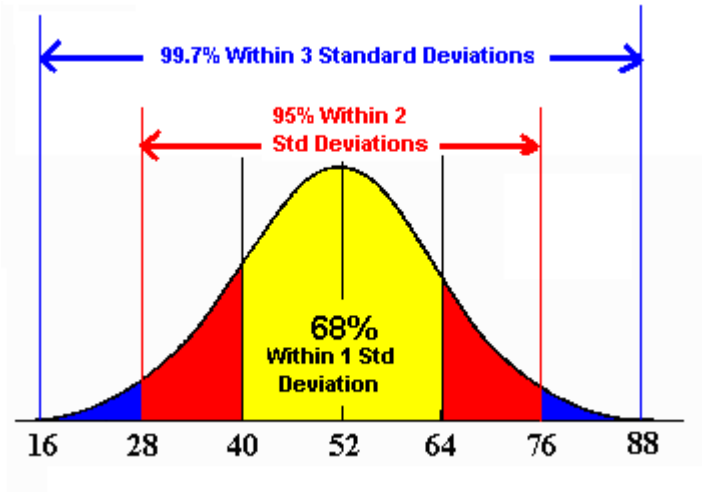
$$deviation = x_i - \bar{x}$$

$$squared\ deviation = (x_i - \bar{x})^2$$

$$avaerage\ squared\ deviation = \frac{\sum(x_i - \bar{x})^2}{n}$$

$$standard\ deviation = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n}}$$

In a normal distribution 68% of the data lies within 1 standard deviation from the mean, 95% within 2 standard deviations, and 99.7% within 3 standard deviations.



When you use a sample, sample's standard deviation tends to be lower than population's standard deviation since sample may not spread out as the population.

Bessel's Correction corrects the bias in the estimation of the population variance, and some (but not all) of the bias in the estimation of the population standard deviation. To apply Bessel's correction, use (n - 1) instead of n. Resulting s is called sample standard deviation.

$$s = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n-1}}$$

Use Bessel's correction primarily to estimate the population standard deviation.

STANDARDIZING

Proportion of data values less than or greater than a certain value in the data set can give a better picture of where that data point stands. Use relative frequencies.

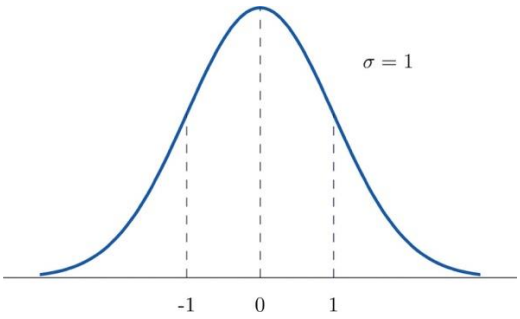
In a distribution with relative frequencies, area under the curve is 1.

When you keep on decreasing the bin size, shape of the distribution is lost and finally end up with a distribution with frequency 1 or 0.

$$Z = \frac{x - \mu}{\sigma}$$

Z score is the number of Standard deviations a value is away from the mean.

Standard normal curve is the resulting distribution we get when we standardize our scores. In order to compare different distributions, we need to standardize distributions using 0 as reference point. (shift the curve to zero by subtracting mean) This is done by converting values to z scores. In a standardized distribution mean is zero and standard deviation is 1.



NORMAL DISTRIBUTION

Probability Distribution Function (PDF) is a curve with an area of 1 beneath it, to represent the cumulative frequency of values.

In a Normal PDF tails never touch x-axis, they get closer and closer to it. X-axis is a horizontal asymptote. An asymptote is a line that a graph approaches but never actually touches.

Area under the curve from negative infinity to x is equal to the probability of randomly selecting a subject in our sample less than x. This is equal to the proportion in the sample or population with scores less than x. If this probability is 80%, x is called the 80th percentile; if it's 90%, it's called 90th percentile.

z-table tabulates z scores & proportion less than those z scores. It's for the standard normal distribution.

Standard Normal Probabilities

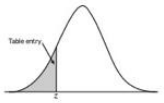


Table entry for z is the area under the standard normal curve to the left of z.

z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
-3.4	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0002
-3.3	.0005	.0005	.0005	.0004	.0004	.0004	.0004	.0004	.0004	.0003
-3.2	.0007	.0007	.0006	.0006	.0006	.0006	.0006	.0005	.0005	.0005
-3.1	.0010	.0009	.0009	.0009	.0008	.0008	.0008	.0008	.0007	.0007
-3.0	.0013	.0013	.0013	.0012	.0012	.0011	.0011	.0011	.0010	.0010
-2.9	.0019	.0018	.0018	.0017	.0016	.0016	.0015	.0015	.0014	.0014
-2.8	.0026	.0025	.0024	.0023	.0023	.0022	.0021	.0021	.0020	.0019
-2.7	.0035	.0034	.0033	.0032	.0031	.0030	.0029	.0028	.0027	.0026
-2.6	.0047	.0045	.0044	.0043	.0041	.0040	.0039	.0038	.0037	.0036
-2.5	.0062	.0060	.0059	.0057	.0055	.0054	.0052	.0051	.0049	.0048
-2.4	.0082	.0080	.0078	.0075	.0073	.0071	.0069	.0068	.0066	.0064
-2.3	.0107	.0104	.0102	.0099	.0096	.0094	.0091	.0089	.0087	.0084
-2.2	.0139	.0136	.0132	.0129	.0125	.0122	.0119	.0116	.0113	.0110
-2.1	.0179	.0174	.0170	.0166	.0162	.0158	.0154	.0150	.0146	.0143
-2.0	.0228	.0222	.0217	.0212	.0207	.0202	.0197	.0192	.0188	.0183
-1.9	.0287	.0281	.0274	.0268	.0262	.0256	.0250	.0244	.0239	.0233
-1.8	.0359	.0351	.0344	.0336	.0329	.0322	.0314	.0307	.0301	.0294
-1.7	.0446	.0436	.0427	.0418	.0409	.0401	.0392	.0384	.0375	.0367
-1.6	.0548	.0537	.0526	.0516	.0505	.0495	.0485	.0475	.0465	.0455
-1.5	.0668	.0655	.0643	.0630	.0618	.0606	.0594	.0582	.0571	.0559
-1.4	.0808	.0793	.0778	.0764	.0749	.0735	.0721	.0708	.0694	.0681
-1.3	.0968	.0951	.0934	.0918	.0901	.0885	.0869	.0853	.0838	.0823
-1.2	.1151	.1131	.1112	.1093	.1075	.1056	.1038	.1020	.1003	.0985
-1.1	.1357	.1335	.1314	.1292	.1271	.1251	.1230	.1210	.1190	.1170
-1.0	.1587	.1562	.1539	.1515	.1492	.1469	.1446	.1423	.1401	.1379
-0.9	.1841	.1814	.1788	.1762	.1736	.1711	.1685	.1660	.1635	.1611
-0.8	.2119	.2090	.2061	.2033	.2005	.1977	.1949	.1922	.1894	.1867
-0.7	.2420	.2389	.2358	.2327	.2296	.2266	.2236	.2206	.2177	.2148
-0.6	.2743	.2709	.2676	.2643	.2611	.2578	.2546	.2514	.2483	.2451
-0.5	.3085	.3050	.3015	.2981	.2946	.2912	.2877	.2843	.2810	.2776
-0.4	.3446	.3409	.3372	.3336	.3300	.3264	.3228	.3192	.3156	.3121
-0.3	.3821	.3783	.3745	.3707	.3669	.3632	.3594	.3557	.3520	.3483
-0.2	.4207	.4168	.4129	.4090	.4052	.4013	.3974	.3936	.3897	.3859
-0.1	.4602	.4562	.4522	.4483	.4443	.4404	.4364	.4325	.4286	.4247
-0.0	.5000	.4960	.4920	.4880	.4840	.4801	.4761	.4721	.4681	.4641

SAMPLING DISTRIBUTIONS

In order to compare multiple samples of the same population, we can compare means of the samples

Sampling distribution is the distribution of sample means. Whatever the shape of the population, sampling distribution is normal. distribution of sample means will become increasingly more normal as the sample size, n, increases.

central limit theorem (CLT) states that given a sufficiently large sample size from a population with a finite level of variance, the mean of all possible samples from the population will be equal to the mean of the population.

standard deviation of the sample means is aka Standard Error(SE) of the population mean.

As sample size increases SE decreases, shape of the sampling distribution gets skinnier.

$$\sigma(\bar{x}) = \frac{\sigma}{\sqrt{n}} \qquad SE = \frac{\sigma}{\sqrt{n}}$$

← Standard deviation

← Number of samples

PYTHON

Python is case sensitive.

Variables

```
mv_population = 74728
```

x = 3 y = 4 z = 5	=	x, y, z = x, 3, 4, 5

Only use ordinary letters, numbers and underscores in variable names. They can't have spaces, and need to start with a letter or underscore.

pythonic way to name variables is to use all lowercase letters and underscores to separate words. It is called snake case, because we tend to connect the words with underscores.

You can't use reserved words or built-in identifiers

Keywords in Python programming language

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

A method in Python behaves similar to a function. Methods are functions that are called using dot notation. Methods are specific to the data type for a particular variable.

there are two types of errors

1. Exceptions
2. Syntax

An Exception is a problem that occurs when the code is running, but a 'Syntax Error' is a problem detected when Python checks the code before it runs it.

DATA TYPES & OPERATORS

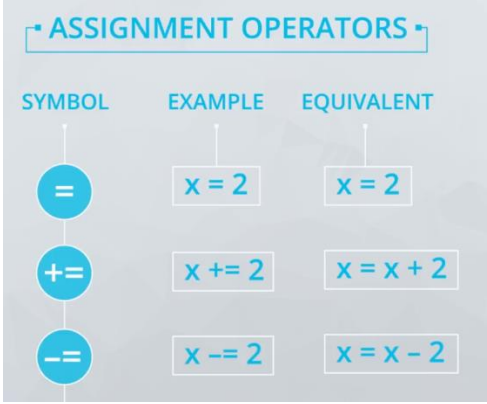
- Operators: Arithmetic, Assignment, Comparison, Logical, Membership, Identity
- Data Types: Integers, Floats, Booleans, Strings, Lists, Tuples, Sets, Dictionaries

ARITHMETIC OPERATORS

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Mod (modulo, remainder after dividing)
**	Exponentiation
//	Integer division, Divides and rounds down to the nearest integer

Usual order of mathematical operations holds in Python.

ASSIGNMENT OPERATORS



COMPARISON OPERATORS

Symbol Use Case	Bool	Operation
5 < 3	False	Less Than
5 > 3	True	Greater Than
3 <= 3	True	Less Than or Equal To
3 >= 5	False	Greater Than or Equal To
3 == 5	False	Equal To
3 != 5	True	Not Equal To

LOGICAL OPERATORS

Logical Use	Bool	Operation
5 < 3 `and` 5 == 5	False	`and` - Evaluates if all provided statements are True
5 < 3 `or` 5 == 5	True	`or` - Evaluates if at least one of many statements is True
`not` 5 < 3	True	`not` - Flips the Bool Value

INTEGERS AND FLOATS

There are two Python data types that could be used for numeric values:

1. int - for integer values
2. float - for decimal or floating point values

You can check the type by using the type function:

```
>>> print(type(633))
int
```

```
>>> print(type("633"))
str
>>> print(type(633.0))
float
```

Whole numbers are automatically integers and numbers with a decimal point are automatically floats. In order for it to be otherwise need to convert manually. Converting a fraction to int will cut off the fractions without rounding it.

```
x = int(4.7)   # x is now an integer 4
y = float(4)   # y is now a float of 4.0
```

An operation involving an int & a float always produces a float.

Because the float, or approximation, for 0.1 is actually slightly more than 0.1, when we add several of them together we can see the difference between the mathematically correct answer and the one that Python creates.

```
>>> print(.1 + .1 + .1 == .3)
False
```

BOOLEAN

The bool data type holds one of the values True or False, which are often encoded as 1 or 0, respectively.

STRING

Strings in Python are shown as the variable type str. You can define a string with either double quotes " or single quotes '. In order to have a quote symbol within the string need to use a \ in front of it. (If you are using single quotes to define a string you can have double quotes within the string without \ & vice versa)

You can index into strings. Python uses 0 indexing.

```
>>> first_word = 'Hello'
>>> second_word = 'There'
>>> print(first_word + second_word)
HelloThere
>>> print(first_word + ' ' + second_word)
Hello There
>>> print(first_word * 5)
HelloHelloHelloHelloHello
>>> print(len(first_word))
5
>>> first_word[0]
H
>>> first_word[1]
e
```

```
capitalize()    encode()        format()
casefold()      endswith()      format_map()
center()        expandtabs()    index()
count()         find()          isalnum()
isalpha()       islower()       istitle()
isdecimal()     isnumeric()     isupper()
isdigit()       isprintable()   join()
isidentifier()  isspace()       ljust()
```

LISTS

List is a mutable, ordered sequence of elements. They can contain any mix and match of the data types. All ordered containers (like lists) follow zero based indexing. You can also index from the end of a list by using negative values, where -1 is the last element, -2 is the second to last element and so on.

we can retrieve subsequence of a list by using slicing. When using slicing, lower index is inclusive and upper index is exclusive. It returns a list. In order to start at the beginning of the list or end at the end of the list, leave start element or end element respectively.

Indexing & slicing works the same on strings, where the returned value will be a string.

```
list_of_random_things = [1, 3.4, 'a string', True]

>>> list_of_random_things[0]
1
>>> list_of_random_things[-1]
True
>>> list_of_random_things[-2]
a string
>>> list_of_random_things[1:2]
[3.4]
>>> list_of_random_things[:2]
[1, 3.4]
>>> list_of_random_things[1:]
[3.4, 'a string', True]
```

we can use in and not in to return a bool of whether an element exists within our list, or if one string is a substring of another.

```
>>> 'this' in 'this is a string'
True
>>> 'in' in 'this is a string'
True
>>> 'isa' in 'this is a string'
False
>>> 5 not in [1, 2, 3, 4, 6]
True
>>> 5 in [1, 2, 3, 4, 6]
False
```

Mutability is about whether or not we can change an object once it has been created. If an object can be changed it is mutable, if not it is immutable. lists are mutable & strings are immutable.

Both strings and lists are ordered.

List functions

len()	Returns how many elements are in a list.
max()	returns the greatest element of the list.
min()	returns the smallest element in a list.
sorted()	returns a copy of a list ordered from smallest to largest
join	Join is a string method that takes a list of strings as an argument, and returns a string consisting of the list elements joined by a separator string. If there are non-string values in the list it will return an error
append	adds an element to the end of a list.
sum	returns the sum of the elements in a list
pop	removes the last element from a list and returns it

*max(), min() are defined using comparison operator. They are undefined for lists that contain elements from different, incomparable types.

```
new_str = "\n".join(["fore", "aft", "starboard", "port"])
print(new_str)
fore
aft
starboard
port

letters = ['a', 'b', 'c', 'd']
letters.append('z')
print(letters)
['a', 'b', 'c', 'd', 'z']
```

TUPLES

A tuple is a data type for immutable ordered sequences of elements. They are often used to store related pieces of information.

similar to lists tuples store an ordered collection of objects which can be accessed by their indices. But tuples are immutable - you can't add and remove items from them, or sort them.

Tuples can also be used to assign multiple variables in a compact way. This is called tuple unpacking.

parentheses are optional when defining tuples

```
location = (13.4125, 103.866667)
print("Latitude:", location[0])
print("Longitude:", location[1])

dimensions = 52, 40, 100
length, width, height = dimensions

length, width, height = 52, 40, 100
```

SETS

A set is a data type for mutable unordered collections of unique elements. One application of a set is to quickly remove duplicates from a list.

Like lists sets support the in operator. You can add elements to sets using add method, and remove elements using pop method, similar to lists. Although, when you pop an element from a set, a random element is removed.

Other operations you can perform with sets include those of mathematical sets. Methods like union, intersection, and difference are easy to perform with sets, and are much faster than such operators with other containers.

```
numbers = [1, 2, 6, 3, 1, 1, 6]
unique_nums = set(numbers)
print(unique_nums)
{1, 2, 3, 6}

fruit = {"apple", "banana", "orange", "grapefruit"}
print("watermelon" in fruit)  # check for element
fruit.add("watermelon")  # add an element
print(fruit)
print(fruit.pop())  # remove a random element
print(fruit)

False
{'grapefruit', 'orange', 'watermelon', 'banana', 'apple'}
grapefruit
{'orange', 'watermelon', 'banana', 'apple'}
```

DICTIONARIES

A dictionary is a mutable data type that stores mappings of unique keys to values.

Dictionaries can have keys of any immutable type, like integers or tuples, not just strings. It's not even necessary for every key to have the same type! We can look up values or insert new values in the dictionary using square brackets that enclose the key

We can check whether a value is in a dictionary with the in keyword. Dicts have a related method that's also useful, get. get looks up values in a dictionary, but unlike square brackets, get returns None (or a default value of your choice) if the key isn't found.

```
elements = {"hydrogen": 1, "helium": 2, "carbon": 6}
elements["lithium"] = 3

print(elements)
{"hydrogen": 1, "helium": 2, "carbon": 6, " lithium ": 3}

print(elements["helium"])
2

print(elements["dilithium "])
KeyError : ["dilithium "])
```

```
print("carbon" in elements)
True

print(elements.get("dilithium"))
None

Print(elements.get('kryptonite', ‘ no such element!'))
"no such element!"
```

IDENTITY OPERATORS

is	evaluates if both sides have the same identity
is not	evaluates if both sides have different identities

You can check if a key returned None with the is operator. You can check for the opposite using is not.

```
n = elements.get("dilithium")
print(n is None)
print(n is not None)
True
False

a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a == b)
print(a is b)
print(a == c)
print(a is c)

True
True
True
False
```

COMPOUND DATA STRUCTURES

We can include containers in other containers to create compound data structures.

```
elements = {    "hydrogen": {   "number": 1,
                                "weight": 1.00794,
                                "symbol": "H"},
                "helium": {     "number": 2,
                                "weight": 4.002602,
                                "symbol": "He"}}

# get the helium dictionary
helium = elements["helium"]

# get hydrogen's weight
hydrogen_weight = elements["hydrogen"]["weight"]
```

SUMMARIZED

[1, 2, 3, 4]	list
(1, 2, 3, 4)	tuple
1, 2, 3, 4	tuple
{1, 2, 3, 4}	set

	list	tuple	set	dictionary
ordered	Y	Y	N	N
mutable	Y	N	Y	Y
unique	N	N	Y	Y

CONTROL FLOW

Indentation

Some languages use braces to show where blocks of code begin and end. In Python indentation is used to enclose blocks of code. Indents conventionally come in multiples of four spaces. (not tab) Be strict about this, because changing the indentation can completely change the meaning of code. Python 3 disallows mixing the use of tabs and spaces for indentation.

CONDITIONAL STATEMENTS

IF STATEMENT

```
if phone_balance < 5:
    phone_balance += 10
    bank_balance -= 10
if season == 'spring':
    print('plant the garden!')
elif season == 'summer':
    print('water the garden!')
elif season == 'fall':
    print('harvest the garden!')
elif season == 'winter':
    print('stay indoors!')
else:
    print('unrecognized season')

if (not unsubscribed) and (location == "USA" or location ==
"CAN"):
    print("send email")
```

- Don't use True or False as conditions
- Be careful writing expressions that use logical operators
- Don't compare a boolean variable with == True or == False

If we use a non-boolean object as a condition in an if statement, Python will check for its truth value and use that to decide whether or not to run the indented code. By default, the truth value of an object in Python is True unless specified as False in the documentation.

Following are considered False in Python:

- constants defined to be false: None and False
- zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)

- empty sequences and collections: '""', (), [], {}, set(), range(0)

FOR LOOPS

A for loop is used to iterate over an iterable.

An iterable is an object that can return one of its elements at a time. This can include sequence types, such as strings, lists, and tuples, as well as non-sequence types, such as dictionaries and files.

We can extract information from lists, create and modify lists with for loops.

range() is a built-in function used to create an iterable sequence of numbers. You can use range() with a for loop to repeat an action a certain number of times. It takes three integer arguments, the first and third of which are optional

range(start=0, stop, step=1)

- 'start': first number of the sequence. If unspecified, defaults to 0.
- 'stop': 1 more than the last number of the sequence. must be specified.
- 'step': difference between each number in the sequence. If unspecified, defaults to 1.

If you specify one integer it's used as the value for 'stop,' and the defaults are used for the other two.

If you specify two integers inside the parentheses with range(), they're used for 'start' and 'stop,' and the default is used for 'step.'

Or you can specify all three integers for 'start', 'stop', and 'step.'

```
range(4) returns 0, 1, 2, 3
range(2, 6) returns 2, 3, 4, 5
range(1, 10, 2) returns 1, 3, 5, 7, 9

cities = ['new york city', 'mountain view', 'chicago']
capitalized_cities = []

for city in cities:
  print(city)

for city in cities:
  capitalized_cities.append(city.title())

for i in range(3):
  print("Hello!")

for index in range(len(cities)):
  cities[index] = cities[index].title()
```

When you iterate through a dictionary using a for loop, doing it the normal way will only give you access to the keys. If you wish to iterate through both keys and values, you can use the built-in method items. Items() returns tuples of key, value pairs, which you can use to iterate over dictionaries in for loops.

```
cast = {
          "Jerry Seinfeld": "Jerry Seinfeld",
          "Julia Louis-Dreyfus": "Elaine Benes",
          "Jason Alexander": "George Costanza",
          "Michael Richards": "Cosmo Kramer"
        }

for key in cast:
  print(key)

Jerry Seinfeld
Julia Louis-Dreyfus
Jason Alexander
Michael Richards

for key, value in cast.items():
  print("Actor: {}   Role: {}".format(key, value))

Actor: Jerry Seinfeld   Role: Jerry Seinfeld
Actor: Julia Louis-Dreyfus   Role: Elaine Benes
Actor: Jason Alexander   Role: George Costanza
Actor: Michael Richards   Role: Cosmo Kramer
```

WHILE LOOPS

For loops are an example of "definite iteration" meaning that the loop's body is run a predefined number of times. This differs from "indefinite iteration" which is when a loop repeats an unknown number of times and ends when some condition is met, which is what happens in a while loop.

```
while sum(hand)  < 17:
  hand.append(card_deck.pop())
```

break and continue keywords can be used in both for and while loops.

- break terminates a loop
- continue skips one iteration of a loop

ZIP

zip returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables.

Like we did for range() we need to convert it to a list or iterate through it with a loop to see the elements. You could unpack each tuple in a for loop.

In addition to zipping two lists together, you can also unzip a list using an asterisk. This would create the same letters and nums lists we saw earlier.

```
list(zip(['a', 'b', 'c'], [1, 2, 3]))
[('a', 1), ('b', 2), ('c', 3)].

letters = ['a', 'b', 'c']
nums = [1, 2, 3]
for letter, num in zip(letters, nums):
   print("{}: {}".format(letter, num))

some_list = [('a', 1), ('b', 2), ('c', 3)]
letters, nums = zip(*some_list)
```

ENUMERATE

enumerate returns an iterator of tuples containing indices and values of a list. use this when you want the index along with each element of an iterable in a loop.

```
letters = ['a', 'b', 'c', 'd', 'e']
for i, letter in enumerate(letters):
   print(i, letter)
0 a
1 b
2 c
3 d
4 e
```

LIST COMPREHENSIONS

In Python, you can create lists really quickly and concisely with list comprehensions. They allow us to create a list using a for loop in one step. They are not found in other languages, but are very common in python.

You create a list comprehension with brackets [], including an expression to evaluate for each element in an iterable. This list comprehension above calls city.title() for each element city in cities, to create each element in the new list, capitalized_cities.

```
capitalized_cities = [city.title() for city in cities]
```

You can also add conditionals to list comprehensions (listcomps). After the iterable, you can use the if keyword to check a condition in each iteration.

```
squares = [x**2 for x in range(9) if x % 2 == 0]
```

Code above sets squares equal to the list [0, 4, 16, 36, 64], as x to the power of 2 is only evaluated if x is even. If you want to add an else, you will get a syntax error doing this.

```
squares = [x**2 for x in range(9) if x % 2 == 0 else x + 3]
```

If you would like to add else, you have to move the conditionals to the beginning of the listcomp, right after the expression, like this.

```
squares = [x**2 if x % 2 == 0 else x + 3 for x in range(9)]
```

FUNCTIONS

Functions allow you to encapsulate a task.

Function Header

- first line of a function definition
- starts with the def keyword
- Then comes the function name, which follows the same naming conventions as variables.
- parentheses that may include arguments separated by commas
- ends with a colon :

Function Body

- body of a function is the code indented after the header line.
- Within body, we can refer to the argument variables and define new variables (local variables), which can only be used within these indented lines
- often includes a return statement. If it doesn't return it will return none by default

We can add default arguments in a function

It is possible to pass values to arguments in two ways - by position and by name. Both are evaluated the same way.

```
def cylinder_volume(height, radius):
   pi = 3.14159
   return height * pi * radius ** 2

cylinder_volume(10, 3)

def cylinder_volume(height, radius=5):
   pi = 3.14159
   return height * pi * radius ** 2

cylinder_volume(10) )  # use default radius 5
cylinder_volume(10, 7) # use 7 as radius

cylinder_volume(10, 7)  # pass in arguments by position
cylinder_volume(height=10, radius=7)  # pass in arguments by name
```

VARIABLE SCOPE

Variable scope refers to which parts of a program a variable can be referenced, or used, from.

If a variable is created inside a function, it can only be used within that function. It is said to have scope that is only local to the function.

A variable defined outside functions can be accessed within a function. It is said to have a global scope. value of a global

variable can not be modified inside the function. If you want to modify that variable's value inside this function, it should be passed in as an argument.

Reusing names for objects is OK as long as you keep them in separate scope. It is best to define variables in the smallest scope they will be needed in.

DOCUMENTATION

Docstrings are a type of comment used to explain the purpose of a function, and how it should be used. Docstrings are surrounded by triple quotes.

```
def population_density(population, land_area):
        """Calculate the population density of an area. """
        return population / land_area

def population_density(population, land_area):
    """Calculate the population density of an area.

    INPUT:
    population: int.
    land_area: int or float.

    OUTPUT:
    population_density: population / land_area.
    """
    return population / land_area
```

LAMBDA EXPRESSIONS

You can use lambda expressions to create anonymous functions. They are helpful for creating quick functions that aren't needed later in your code. This is especially useful for higher order functions, or functions that take in other functions as arguments. They aren't ideal for complex functions, but are useful for short, simple functions.

Components of a Lambda Function

- lambda keyword
- one or more arguments separated by commas
- a colon :
- an expression that is evaluated and returned

```
multiply = lambda x, y: x * y
multiply(4, 7)
```

ITERATORS & GENERATORS

Iterables are objects that can return one of their elements at a time, such as a list. Many of the built-in functions, like 'enumerate,' return an iterator.

An iterator is an object that represents a stream of data. This is different from a list, which is also an iterable, but not an iterator because it is not a stream of data.

Generators are a simple way to create iterators using functions. You can also define iterators using classes.

Following generator function my_range, produces an iterator that is a stream of numbers from 0 to (x - 1).

instead of using return keyword, it uses yield. This allows the function to return values one at a time, and start where it left off each time it's called. This yield keyword is what differentiates a generator from a typical function.

since this returns an iterator, we can convert it to a list or iterate through it in a loop to view its contents.

```
def my_range(x):
        i = 0
        while i < x:
            yield i
            i += 1

for x in my_range(4):
        print(x)

        0
        1
        2
        3
```

Generators are a lazy way to build iterables. They are useful when the fully realized list would not fit in memory, or when the cost to calculate each list element is high and you want to do it as late as possible. But they can only be iterated over once.

Generator Expressions; You can create a generator in the same way you'd normally write a list comprehension, except with parentheses instead of square brackets.

```
sq_list = [x**2 for x in range(10)]
# this produces a list of squares

sq_iterator = (x**2 for x in range(10))
# this produces an iterator of squares
```

SCRIPTING

Anaconda version	conda --version
Python version	python --version
access Python interpreter	python
leave Python interpreter	exit()
Run a Python Script	python first_script.py

SCRIPTING WITH RAW INPUT

We can get raw input from the user with the built-in function input. It has an optional string argument to specify a message to show to the user when asking for input. It takes in whatever the user types and stores it as a string. If

you want to interpret input as something other than a string, wrap the result with the new type to convert it.

```
name = input("Enter your name: ")
print("Hello there, {}!".format(name.title()))

num = int(input("Enter an integer"))
print("hello" * num)
```

We can also interpret user input as a Python expression using the built-in function eval. This evaluates a string as a line of Python.

```
result = eval(input("Enter an expression: "))
print(result)
```

ERRORS AND EXCEPTIONS

Syntax errors occur when Python can't interpret our code, since we didn't follow the correct syntax for Python.

Exceptions occur when unexpected things happen during program execution, even if the code is syntactically correct. There are different types of built-in exceptions, and you can see which exception is thrown in the error message.

We can use try statements to handle exceptions. There are four clauses you can use.

- **try**: This is the only mandatory clause in a try statement. The code in this block is the first thing that Python runs in a try statement.
- **except**: If it runs into an exception while running the try block, it will jump to the except block that handles that exception.
- **else**: If it runs into no exceptions while running the try block, it will run the code in this block after running the try block.
- **finally**: Before Python leaves try statement, it will always run the code in finally block, even if it's ending the program. E.g., if Python ran into an error while running code in the except or else block, this finally block will still be executed before stopping the program.

```
try:
  # some code
except ValueError:
  # some code

try:
  # some code
except ValueError, KeyboardInterrupt:
  # some code

try:
  # some code
except ValueError:
  # some code
except KeyboardInterrupt:
  # some code

try:
  # some code
except ZeroDivisionError as e:
  # some code
  print("ZeroDivisionError occurred: {}".format(e))
```

READING AND WRITING FILES

First open the file using the built-in function, open. This requires path to the file. It returns a file object through which Python interacts with the file itself. There are optional parameters in the open function. One is the mode in which we open the file. (default is read)

Use read() to access content from the file object.

When finished with the file, use close() to free up any system resources taken up by the file.

Open the file in writing ('w') mode. If the file does not exist, it will be created. If you open an existing file in writing mode, its content will be deleted. If you're interested in adding to an existing file, without deleting its content, use append ('a') mode instead of write.

Use write method to add text to the file. Close the file when finished.

```
f = open('my_path/my_file.txt', 'r')
file_data = f.read()
f.close()

f = open('my_path/my_file.txt', 'w')
f.write("Hello there!")
f.close()
```

If you open too many files without closing them, you can run out of file handles & you won't be able to open any new files.

Python provides a special syntax that auto-closes a file for you once you're finished using it. This with keyword allows you to open a file, do operations on it, and automatically close it after the indented code is executed. we don't have to call f.close()! You can access the file object, f, only within this indented block.

```
with open('my_path/my_file.txt', 'r') as f:
  file_data = f.read()
```

By default f.read() will read the remainder of the file from its current position - the whole file. If you pass an integer argument, it will read up to that number of characters, output all of them, and keep the 'window' at that position ready to read on.

```
with open("camelot.txt") as song:
   print(song.read(2))
   print(song.read(8))
   print(song.read())
```

Newline character(\n) marks the end of a line, and tells a program (such as a text editor) to go down to the next line. However, looking at the stream of characters in the file, \n is just another character.

```
f.readline()

camelot_lines = []
with open("camelot.txt") as f:
   for line in f:
      camelot_lines.append(line.strip())
```

IMPORTING LOCAL SCRIPTS

We can import Python code from other scripts, which is helpful if you are working on a bigger project where you want to organize your code into multiple files and reuse code in those files. If the Python script you want to import is in the same directory as your current script, you type import followed by file name, without.py extension.

It's the standard convention for import statements to be written at the top of a Python script, each one on a separate line. This import statement creates a module object called useful_functions. Modules are Python files that contain definitions and statements. To access objects from an imported module, you need to use dot notation. We can add an alias to an imported module to reference it with a different name.

Python Standard Library is organized into modules. Find the documentation at https://docs.python.org/3/library/

modules in the Python Standard Library are split down into sub-modules that are contained within a package. A package is simply a module that contains sub-modules. A sub-module is specified with the usual dot notation.

submodules are specified by the package name and then the submodule name separated by a dot.

```
import useful_functions
# creates a module object called useful_functions
useful_functions.add_five([1, 2, 3, 4])

import useful_functions as uf
uf.add_five([1, 2, 3, 4])

import package_name.submodule_name
```

To import an individual function or class from a module:
from module_name import object_name

To import multiple individual objects from a module:
from module_name import first_object, second_object
To rename a module
import module_name as new_name
To import an object from a module and rename it
from module_name import object_name as new_name
To import every object individually from a module (DO NOT DO THIS):
from module_name import *
If you want to use all of the objects from a module, use the standard import module_name and access each of the objects with the dot notation.
import module_name

USEFUL THIRD-PARTY PACKAGES

There are thousands of third-party libraries written by independent developers! You can install them using pip, a package manager that is included with Python 3. pip is the standard package manager for Python, but it isn't the only one. One popular alternative is Anaconda which is designed specifically for data science.

Following command downloads and installs the package so that it's available to import in your programs. Once installed, you can import third-party packages using the same syntax used to import from the standard library

```
pip install package_name
```

Larger Python programs might depend on dozens of third party packages. To make it easier to share these programs, project's dependencies are listed in a file called requirements.txt.

Each line of the file includes the name of a package and its version number.

```
beautifulsoup4==4.5.1
bs4==0.0.1
pytz==2016.7
requests==2.11.1
```

You can use pip to install all of a project's dependencies at once by following command

```
pip install -r requirements.txt
```

IPython	A better interactive Python interpreter
requests	Provides easy to use methods to make web requests. Useful for accessing web APIs.

Flask	a lightweight framework for making web applications and APIs.
Django	more featureful framework for making web applications. Django is particularly good for designing complex, content heavy, web applications.
Beautiful Soup	Used to parse HTML and extract information from it. Great for web scraping.
pytest	extends Python's builtin assertions and unittest module.
PyYAML	For reading and writing YAML files.
NumPy	fundamental package for scientific computing with Python. It contains among other things a powerful N- dimensional array object and useful linear algebra capabilities.
pandas	A library containing high performance, data structures and data analysis tools. In particular, pandas provides dataframes!
matplotlib	a 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments.
ggplot	Another 2D plotting library, based on R's ggplot2 library.
Pillow	The Python Imaging Library adds image processing capabilities to your Python interpreter.
pyglet	A cross platform application framework intended for game development.
Pygame	A set of Python modules designed for writing games.
pytz	World Timezone Definitions for Python

INTERPRETER

Start python interactive interpreter by entering command python in terminal. You can type here to interact with Python directly. This is a great place to experiment with Python code. Just enter code, and the output will appear on the next line.

In the interpreter, value of the last line in a prompt will be outputted automatically. If you have multiple lines where you want to output values, use print.

If you start to define a function you will see a change in the prompt, to signify that this is a continuation line. Include indentation as you define the function.

A drawback of the interpreter is that it's tricky to edit code. If you made a mistake when typing a function, or forgot to indent the body of the function, you can't use the mouse to click your cursor where you want it. You have to navigate with arrow keys to move the cursor forwards and backwards for editing. It would be helpful to learn shortcuts for actions like moving to the beginning or end of the line.

You can reference any objects you defined earlier in the interpreter!

Use up and down arrow to cycle through your recent commands at the interactive prompt. This can be useful to re-run or adapt code you've already tried.

To quit the Python interactive interpreter, use the command exit() or hit ctrl-D on mac or linux, and ctrl-Z then Enter for windows.

```
>>> type(5.23)
<class 'float'>
>>> def cylinder_volume(height, radius):
...         pi = 3.14159
...         return height * pi * radius ** 2
>>> cylinder_volume(10, 3)
282.7431
```

There is an alternative to the default python interactive interpreter, IPython, which comes with many additional features.

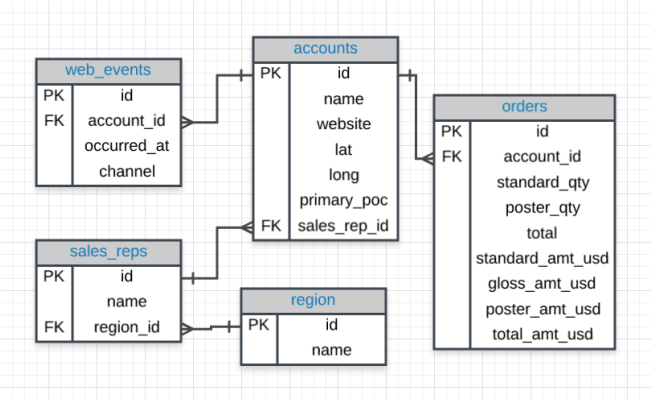
- tab completion
- ? for details about an object
- ! to execute system shell commands
- syntax highlighting!

SQL

BASIC SQL

SQL = STRUCTURED QUERY LANGUAGE

Entity Relationship Diagram (ERD) is a diagram that shows how data is structured in a db.



The "crow's foot" that connects the tables together shows us how the columns in one table relate to the columns in another table.

There are some major advantages to using traditional relational dbs, which we interact with using SQL.

- SQL is easy to understand.
- Traditional dbs allow us to access data directly.
- Traditional dbs allow us to audit and replicate our data.
- SQL is a great tool for analyzing multiple tables at once.
- SQL allows you to analyze more complex questions than dashboard tools like Google Analytics.

Why Businesses Like Databases

Data integrity is ensured - only the data you want is entered & only certain users are able to enter data into the db.

Data can be accessed quickly - SQL allows to obtain results quickly. Code can be optimized to quickly pull results.

Data is easily shared - multiple individuals can access data stored in a db, and the data is the same for all users allowing for consistent results for anyone with access to your db.

All data in a column must be of the same type

SQL Databases

- MySQL
- Access
- Oracle
- Microsoft SQL Server
- Postgres

SQL is not case sensitive.

If using a non-numeric with an operator, must put the value in single quotes.

Creating a new column that is a combination of existing columns is known as a derived column.

order of columns listed in the ORDER BY clause matters. You are ordering the columns from left to right

Statement	How to Use It	Other Details
SELECT	SELECT Col1, Col2, ...	Provide the columns you want
FROM	FROM Table	Provide the table where the columns exist
LIMIT	LIMIT 10	Limits based number of rows returned
ORDER BY	ORDER BY Col	Orders table based on the column. Used with DESC.
WHERE	WHERE Col > 5	A conditional statement to filter your results
LIKE	WHERE Col LIKE '%me%'	Only pulls rows where column has 'me' within the text
IN	WHERE Col IN ('Y', 'N')	A filter for only rows with column of 'Y' or 'N'
NOT	WHERE Col NOT IN ('Y', 'N')	NOT is frequently used with LIKE and IN
AND	WHERE Col1 > 5 AND Col2 < 3	Filter rows where two or more conditions must be true
OR	WHERE Col1 > 5 OR Col2 < 3	Filter rows where at least one condition must be true
BETWEEN	WHERE Col BETWEEN 3 AND 5	Often easier syntax than using an AND

```
SELECT * FROM orders;

SELECT * FROM orders LIMIT 10;

SELECT * FROM orders ORDER BY occurred_at LIMIT 10;

SELECT * FROM orders ORDER BY  occurred_at DESC LIMIT 10;

SELECT * FROM orders ORDER BY  account_id, occurred_at DESC LIMIT 10;

SELECT * FROM orders WHERE account_id = 4251;

SELECT * FROM accounts WHERE name = 'Walmart';

SELECT standard_amt_usd/standard_qty as unit_price FROM orders;

SELECT * FROM accounts WHERE name LIKE '%Ca%'

SELECT * FROM accounts WHERE name IN ('Walmart', 'Target');

SELECT * FROM accounts WHERE name NOT LIKE '%Ca%'

SELECT * FROM accounts WHERE name NOT IN ('Walmart', 'Target');

SELECT * FROM accounts WHERE name NOT  LIKE 'C%' AND name NOT LIKE '%s';

SELECT * FROM orders WHERE occurred_at BETWEEN '2016-01-01' AND '2017-01-01'
```

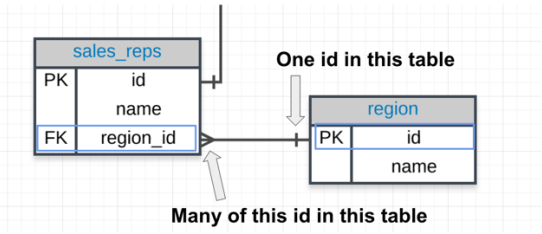
SQL JOINS

Database Normalization

When creating a db, it is really important to think about how data will be stored. This is known as normalization, and it is a huge part of most SQL classes. If you are in charge of setting up a new db, it is important to have a thorough understanding of database normalization.

There are essentially three ideas that are aimed at db normalization:

1. Are the tables storing logical groupings of the data?
2. Can I make changes in a single location, rather than in many tables for the same information?
3. Can I access and manipulate data quickly & efficiently?



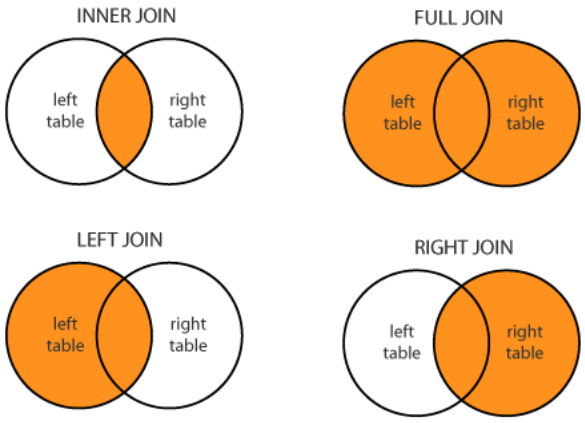
Primary Key (PK)

PK exists in every table, and it is a column that has a unique value for every row. It is common for it to be the first column in tables.

Foreign Key (FK)

FK is linked to the primary key of another table.

Joins



JOINs allow us to pull data from multiple tables.

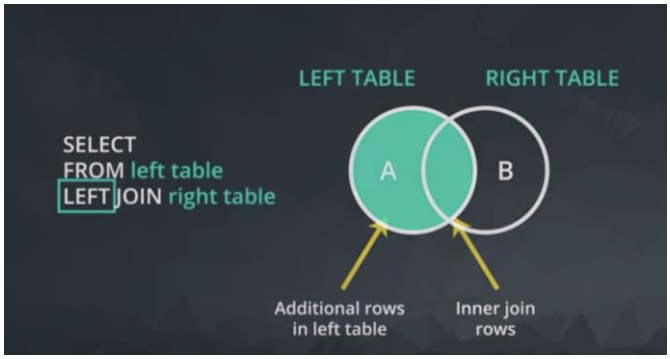
In a inner join, result is the same as if you switched the tables in FROM and JOIN. Which side of the = a column is listed doesn't matter.

In the ON, we will ALWAYs have PK equal to FK.

When we JOIN tables, it is good to give each table an alias. Frequently first letter of table name id used as the alias.

If you have two or more columns in your SELECT that have the same name after the table name such as accounts.name and sales_reps.name you need to alias them. Otherwise it will only show one of the columns.

INNER JOIN always pulls rows only if they exist as a match across two tables. Other JOINs pull rows that might only exist in one of the two tables. This will introduce a new data type called NULL. a full outer join will return the inner join result set, as well as any unmatched rows from either of the two tables being joined.



Left & right joins are effectively interchangeable. (Change from & join)

```
SELECT accounts.name, orders.occurred_at
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;

SELECT *
FROM web_events
JOIN accounts
ON web_events.account_id = accounts.id
JOIN orders
ON accounts.id = orders.account_id

SELECT w.channel, a.name
FROM web_events AS w
JOIN accounts AS a
ON w.account_id=a.id
```

Following 2 are the same

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
LEFT JOIN State s
ON c.countryid = s.countryid;

SELECT c.countryid, c.countryName, s.stateName
FROM State s
RIGHT JOIN Country c
ON c.countryid = s.countryid;
```

Logic in WHERE clause occurs after the join occurs. (First query) Logic in ON clause reduces the rows before combining tables. (Second query)

First query results < second query results

```
SELECT *
FROM orders o
LEFT JOIN accounts a
ON o.account_id = a.id
WHERE a.sales_rep_id = 322

SELECT *
FROM orders o
LEFT JOIN accounts a
ON o.account_id = a.id
AND a.sales_rep_id = 322
```

SQL AGGREGATIONS

COUNT

COUNT can help identify number of null values in a particular column. In the second statement it counts only the rows that has id as not null.

```
SELECT COUNT(*) as order_count
FROM accounts;
SELECT COUNT(accounts.id)
FROM accounts;
```

SUM

Unlike COUNT, you can only use SUM on numeric columns. It ignores NULL values

```
SELECT SUM(poster_qty) AS total_poster_sales
FROM orders;
SELECT SUM(standard_amt_usd)/SUM(standard_qty) AS standard_price_per_unit
FROM orders;
```

MIN and MAX

MIN and MAX can be used on non-numerical columns. Depending on the column type, MIN will return the lowest number, earliest date, or non-numerical value as early in the alphabet as possible. MAX does the opposite. They ignore NULL values.

```
SELECT MIN(occurred_at) FROM orders;
SELECT MAX(occurred_at) FROM web_events;
```

AVG

AVG returns the mean of the data (sum of all of the values in the column divided by the number of values in a column). It ignores NULL values in both numerator & denominator.

```
SELECT AVG(standard_qty) mean_standard
FROM orders;
```

GROUP BY

GROUP BY can be used to aggregate data within subsets of the data. For example, grouping for different accounts, different regions.

Any column in the SELECT statement that is not within an aggregator must be in the GROUP BY clause. (Important)

GROUP BY always goes between WHERE and ORDER BY.

You can GROUP BY multiple columns at once. Order of column names in GROUP BY clause doesn't matter

```
SELECT channel, COUNT(*)
FROM web_events
GROUP BY channel, account_id
ORDER BY channel, account_id
```

DISTINCT

always used in SELECT statements. It provides unique rows for all columns written in the SELECT statement.

```
SELECT DISTINCT column1, column2, column3
FROM table1;
```

HAVING

Where clause doesn't allow you to filter on aggregate columns. any time you want to perform a WHERE on a query that was created by an aggregate, you need to use HAVING instead. This is only useful when grouping by one or more columns since if it's an aggregation on the entire data set, output is one line and it's not required to filter that.

HAVING is the "clean" way to filter a query that has been aggregated, but this is also commonly done using a subquery.

It appears after group by & before order by

```
SELECT s.id, s.name, COUNT(*) num_accounts
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.id, s.name
HAVING COUNT(*) > 5
ORDER BY num_accounts;
```

DATE_TRUNC & DATE_PART

Format of dates in SQL: YYYY-MM-DD HH:MI:SS

Since date related data is very detailed down to a second, data points are unique and is not very useful when aggregating. Date functions to rescue.

DATE_TRUNC allows you to truncate your date to a particular part of your date-time column. Common truncations are day, month, and year.

DATE_PART can be useful for pulling a specific portion of a date, but notice pulling month or day of the week means that you are no longer keeping the years in order. Rather you are grouping for certain components regardless of which year they belonged in.

```
SELECT DATE_PART('year', occurred_at) ord_year,
SUM(total_amt_usd) total_spent
 FROM orders
```

```
 GROUP BY 1
 ORDER BY 2 DESC;

SELECT DATE_PART('dow', occurred_at) as day_of_week,
SUM(total) as total_qty
 FROM orders
 GROUP BY 1
ORDER BY 2 DESC;

SELECT DATE_TRUNC('month', o.occurred_at) ord_date,
SUM(o.gloss_amt_usd) tot_spent
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUP BY 1
```

CASE

CASE statement always goes in the SELECT clause.

CASE must include following components: WHEN, THEN, and END. ELSE is an optional component to catch cases that didn't meet any other CASE conditions.

You can make a conditional statement using any conditional operator (like WHERE) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.

You can include multiple WHEN statements.

```
SELECT account_id,
CASE WHEN standard_qty = 0 OR standard_qty IS NULL
THEN 0
ELSE standard_amt_usd/standard_qty
END AS unit_price
FROM orders
LIMIT 10;

SELECT
CASE WHEN total>500 THEN 'Over 500'
ELSE '500 or under'END AS total_group
COUNT(*) AS order_count
FROM orders
GROUP BY 1
```

Some Key Points

NULLs are different than a zero - they are cells where data does not exist. When identifying NULLs in a WHERE clause, we write IS NULL or IS NOT NULL. We don't use =, because NULL isn't considered a value in SQL.

```
SELECT * FROM accounts WHERE primary_poc IS NULL
```

aggregators only aggregate vertically - the values of a column. If you want to perform a calculation across rows, you can do this with simple arithmetic.

If you want to count NULLs as zero, you will need to use SUM and COUNT. It might not be a good idea if the NULL values represent unknown values for a cell.

SQL evaluates the aggregations before the LIMIT clause.

You can reference the columns in your select statement in GROUP BY and ORDER BY clauses with numbers that follow the order they appear in the select statement.

```
SELECT standard_qty, COUNT(*)
FROM orders
GROUP BY 1
ORDER BY 1
```

SQL SUBQUERIES & TEMPORARY TABLES

Both subqueries and table expressions are ways to write a query that creates a table, and then write a query that interacts with this newly created table.

Subqueries (inner queries/nested queries) are a tool for performing operations in multiple steps. Subquery is a query within a query. Subqueries that return a table (instead of a single value) are required to have aliases.

Inner query will run first & the outer query will run across the result set created by the inner query.

Ensure the query is well formatted since subqueries can make it complex & difficult to read.

if the subquery is returning a single value, you might use that value in a logical statement like WHERE, HAVING, or even SELECT - the value could be nested within a CASE statement.

Most conditional logic will work with Subqueries returning a single value. IN is the only type of conditional logic that will work if Subquery returns multiple results.

You should not include an alias when you write a subquery in a conditional statement. This is because the subquery is treated as an individual value (or set of values in the IN) rather than as a table.

```
SELECT channel, AVG(events) AS average_events
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
        channel, COUNT(*) as events
        FROM web_events
        GROUP BY 1,2) sub
GROUP BY channel
ORDER BY 2 DESC

SELECT SUM(total_amt_usd)
FROM orders
WHERE DATE_TRUNC('month', occurred_at) =
```

```
    (SELECT DATE_TRUNC('month', MIN(occurred_at))
    FROM orders);
```

Common Table Expression

WITH statement is often called a Common Table Expression or CTE. Subqueries can make your queries lengthy & difficult to read. CTEs can help break the query into separate components so that the logic is more readable.

You can have as many CTEs as you want. They need to be defined at the beginning of query. Each CTE gets an alias.

If you have a CTE/subquery that takes a long time to run, you can run it separately & write it back to the DB as it's own table.

```
WITH events AS (
        SELECT DATE_TRUNC('day',occurred_at) AS day,
                channel, COUNT(*) as events
        FROM web_events
        GROUP BY 1,2)
SELECT channel, AVG(events) AS average_events
FROM events
GROUP BY channel
ORDER BY 2 DESC;

WITH table1 AS (
        SELECT *
        FROM web_events),

        table2 AS (
        SELECT *
        FROM accounts)
SELECT *
FROM table1
JOIN table2
ON table1.account_id = table2.id;
```

SQL DATA CLEANING

LEFT
pulls a specified number of characters for each row in a specified column starting at the beginning (from left). LEFT(phone_number, 3)
RIGHT
pulls a specified number of characters for each row in a specified column starting at the end (from right). RIGHT(phone_number, 8)
LENGTH
provides the number of characters for each row of a specified column. LENGTH(phone_number)
POSITION
takes a character and a column, and provides the index where that character is for each row. index of the first position is 1 in SQL. POSITION(',' IN city_state).
STRPOS
STRPOS provides the same result as POSITION, but the syntax is a bit different STRPOS(city_state, ',').
LOWER , UPPER
both POSITION and STRPOS are case sensitive, so looking for A is different than looking for a. If you want to pull an index regardless of the case of a letter, you can use LOWER or UPPER to make all of the characters lower or uppercase.
CONCAT, Piping ||
allow you to combine columns together. CONCAT(first_name, ' ', last_name) first_name || ' ' || last_name.
TO_DATE
TO_DATE function converts a string to a date. Following converts a month name into the number associated with that particular month. TO_DATE('May', 'month') Result: 5 TO_DATE('2003/07/09', 'yyyy/mm/dd') Result: date value of July 9, 2003
CAST
CAST is useful to change column types, like converting strings into numbers & dates. you can change a string to a date using CAST(date_column AS DATE). However, you might want to make other changes to your columns in terms of their data types. Get the string into the correct format first & use CAST to tell it's in fact a date. SELECT date orig_date, CAST((SUBSTR(date, 7, 4) || '-' || LEFT(date, 2) || '-' || SUBSTR(date, 4, 2)) as DATE) new_date FROM sf_crime_data
Casting with ::

instead of CAST(date_column AS DATE), you can also use date_column::DATE. SELECT date orig_date, (SUBSTR(date, 7, 4) || '-' || LEFT(date, 2) || '-' || SUBSTR(date, 4, 2))::DATE new_date FROM sf_crime_data;
TRIM
can be used to remove characters from the beginning and end of a string. This can remove unwanted spaces at the beginning or end of a row that often happen with data being moved from Excel or other storage systems.
COALESCE
COALESCE returns the first non-NULL value in a list. This can be used to replace the nulls. SELECT COALESCE(NULL, NULL, NULL, 'W3Schools.com', NULL, 'Example.com'); Returns: W3Schools.com SELECT * , COALESCE(primary_poc,'no_POC') AS modified FROM accounts Returns: primary_poc if it's not null, else it will return 'no_POC'

LEFT, RIGHT, and TRIM are all used to select only certain elements of strings, but using them to select elements of a number or date will treat them as strings for the purpose of the function. If you want to convert a number in to String perform any type of String operation on it.

SUBSTR function extracts a substring from a string (starting at any position).

SUBSTR(string, start, length)