Learn the Command Line : Navigating the File System

codecademy

The Command Line

The *command line* allows a user to navigate the filesystem and run built-in programs or custom scripts. In Unix, the command line interface is called Bash, and the shell prompt is the `$`.

```
$
```

`ls` List

The shell command `ls` is used to list the contents of directories. If no arguments are given, it will list the contents of the current working directory.

```
$ ls Desktop
resume.pdf
photo.png
```

`pwd` Print Working Directory

The shell command `pwd` displays the file path from the root directory to the current working directory.

```
$ pwd
/Users/sonny/Downloads
```

`cd` Change Directory

The shell command `cd` can be used to move throughout the filesystem of a computer. It accepts a variety of arguments:

- Full file paths.
- Names of children of the current directory.
- `..` the parent of the current directory.

```
$ cd some-directory
$ cd ..
```

`mkdir` Make Directory

The shell command `mkdir` can be used to make a new directory in the filesystem according to its argument. If a file path is given, the new directory will be placed at the end. Otherwise, it will create a new directory in the current working directory with the name given.

```
$ mkdir new-directory
$ ls
old-directory    new-directory
```

`touch` Create New File

The shell command `touch` creates a new file in the current working directory with the name provided.

```
$ touch secret-file.txt
```

Introduction to HTML : Elements and Structure

codecademy

`<table>` Table Element

In HTML, the `<table>` element has content that is used to represent a two-dimensional table made of rows and columns.

```
<table>
  <!-- rows and columns will go here -->
</table>
```

`<tr>` Table Row Element

In HTML, the table row element, `<tr>`, is used to add rows to a table before adding table data and table headings.

```
<table>
  <tr>
    ...
        </tr>
</table>
```

`<td>` Table Data Element

The table data element, `<td>`, can be nested inside a table row element, `<tr>`, to add a cell of data to a table.

```
<table>
  <tr>
        <td>cell one data</td>
        <td>cell two data</td>
        </tr>
</table>
```

`colspan` Attribute

The `colspan` attribute on a table header `<th>` or table data `<td>` element indicates how many columns that particular cell should span within the table. The `colspan` value is set to 1 by default and will take any positive integer between 1 and 1000.

```
<table>
  <tr>
    <th>row 1:</th>
    <td>col 1</td>
    <td>col 2</td>
    <td>col 3</td>
  </tr>
  <tr>
    <th>row 2:</th>
    <td colspan="2">col 1 (will span 2 columns)</td>
    <td>col 2</td>
    <td>col 3</td>
  </tr>
</table>
```

`rowspan` Attribute

Similar to `colspan`, the `rowspan` attribute on a table header or table data element indicates how many rows that particular cell should span within the table. The `rowspan` value is set to 1 by default and will take any positive integer up 65534.

```
<table>
  <tr>
    <th>row 1:</th>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
  <tr>
    <th rowspan="2">row 2 (this row will span 2
rows):</th>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
  <tr>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
   <tr>
    <th>row 3:</th>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
</table>
```

`<tbody>` Table Body Element

The table body element, `<tbody>`, is a semantic element that will contain all table data other than table heading and table footer content. If used, `<tbody>` will contain all table row `<tr>` elements, and indicates that `<tr>` elements make up the body of the table. `<table>` cannot have both `<tbody>` and `<tr>` as direct children.

```
<table>
 <tbody>
   <tr>
     <td>row 1</td>
   </tr>
   <tr>
     <td>row 2</td>
   </tr>
   <tr>
     <td>row 3</td>
   </tr>
 </tbody>
</table>
```

`<thead>` Table Head Element

The table head element, `<thead>`, defines the headings of table columns encapsulated in table rows.

```
<table>
 <thead>
   <tr>
     <th>heading 1</th>
     <th>heading 2</th>
     <th>heading 3</th>
   </tr>
 </thead>
 <tbody>
   <tr>
     <td>col 1</td>
     <td>col 2</td>
     <td>col 3</td>
   </tr>
 </tbody>
</table>
```

`<tfoot>` Table Footer Element

The table footer element, `<tfoot>`, uses table rows to give footer content or to summarize content at the end of a table.

```
<table>
 <thead>
   <tr>
     <th>heading 1</th>
     <th>heading 2</th>
     <th>heading 3</th>
   </tr>
 </thead>
 <tbody>
   <tr>
     <td>col 1</td>
     <td>col 2</td>
     <td>col 3</td>
   </tr>
 </tbody>
 <tfoot>
   <tr>
     <td>summary of col 1</td>
     <td>summary of col 2</td>
     <td>summary of col 3</td>
   </tr>
 </tfoot>
</table>
```

`<th>` Table Heading Element

In HTML, the table heading element, `<th>`, is used to add titles to rows and columns of a table and must be enclosed in a table row element, `<tr>`.

```
<table>
  <tr>
        <th>column one</th>
        <th>column two</th>
        <tr>
        <td>1</td>
                <td>2</td>
        </tr>
</table>
```

Introduction to HTML : Elements and Structure

HTML Element Definition and Syntax

An HTML element is a piece of content in an HTML document and uses the following syntax: opening tag + content + closing tag. In the code provided, `<p>` is the opening tag, `Hello World!` is the content, and `</p>` is the closing tag.

```
<p>Hello World!</p>
```

HTML Tag Syntax

The syntax for a single HTML tag is an opening angle bracket (`<`) followed by the element name and closed with the closing angle bracket (`>`). The above code block is an example of a single opening `<div>` tag.

```
<div>
```

Element Content

The content of an HTML element is the information between the opening and closing tags of an element. In the example, the element content is “Codecademy is awesome!”.

```
<h1>Codecademy is awesome!</h1>
```

Closing Tag

An HTML closing tag is used to denote the end of an HTML element. The syntax for a closing tag is a left angle bracket `<` followed by a forward slash `/` then the element name and a right angle bracket to close `>`.

```
<body>
  ...
</body>
```

Unique ID Attributes

In HTML, specific and unique `id` attributes can be assigned to different elements in order to differentiate between them.

When needed, the `id` value can be called upon by CSS and JavaScript to manipulate, format, and perform specific instructions on that element and that element only. Valid `id` attributes should begin with a letter and should only contain letters (`a-Z`), digits (`0-9`), hyphens (`-`), underscores (`_`), and periods (`.`).

```
<h1 id="A1">Hello World</h1>
```

`<body>` Body Element

The `<body>` element represents the content of an HTML document. Content inside `<body>` tags are rendered on the web browsers.

```
<body>
  <h1>Learn to code with Codecademy :)</h1>
</body>
```

HTML Structure

HTML is organized into a family tree structure. HTML elements can have parents, grandparents, siblings, children, grandchildren, etc.

```
<body>
  <div>
    <h1>A grandchild of body and a child of div</h1>
    <h2>A sibling of h1</h2>
  </div>
</body>
```

`<h1>` `<h2>` `<h3>` `<h4>` `<h5>` `<h6>` Headings

HTML can use six different levels of heading elements. The heading elements are ordered from the highest level `<h1>` to the lowest level `<h6>`.

```
<h1>Breaking News</h1>
<h2>This is the first subheading</h2>
<h3>This is the second subheading</h3>
...
<h6>This is the fifth subheading</h6>
```

`<div>` Div Element

The `<div>` element is used as a container that divides an HTML document into sections and is short for “division”. `<div>` elements can contain *flow content* such as headings, paragraphs, links, images, etc.

```
<div>
  <h1>This is a section of grouped elements</h1>
  <p>Here’s some text for the first section</p>
</div>
<div>
  <h1>This is a second section</h1>
  <p>Here’s some text for the second section</p>
</div>
```

`<ul>` Unordered List Element

The `<ul>` unordered list element is used to create a list of items in no particular order. Each individual list item will have a bullet point by default. For example:

```
<ul>
  <li>Oculus Quest</li>
  <li>Fender Telecaster</li>
  <li>Record Player</li>
</ul>
```

- Oculus Quest
- Fender Telecaster
- Record Player

`<p>` Paragraph Element

Paragraph elements, `<p>`, contain and display blocks of text.

```
<p>This is a block of text! Lorem ipsum dolor sit amet, consectetur adipisicing elit.</p>
```

`<span>` Element

The `<span>` element is an inline container for text and can be used to group text for styling purposes. However, as `<span>` is a generic container to separate pieces of text from a larger body of text, its use should be avoided if a more semantic element is available.

```
<p><span>This text</span> may be styled differently than the surrounding text.</p>
```

`<em>` Emphasis Element

The `<em>` emphasis element emphasizes text and browsers will usually *italicize* the emphasized text by default.

```
<p>This <em>word</em> will be emphasized in italics.
</p>
```

`<strong>` Strong Element

The `<strong>` element highlights important, serious, or urgent text and browsers will normally render this highlighted text in **bold** by default.

```
<p>This is <strong>important</strong> text!</p>
```

`<br>` Line Break Element

The `<br>` line break element will create a line break in text and is especially useful where a division of text is required, like in a postal address. The line break element requires only an opening tag and must not have a closing tag.

```
<p>A line break haiku. <br>Poems are a great use case. <br>Oh joy! A line break.</p>
```

`<li>` List Item Element

The `<li>` list item element create list items inside:

- *Ordered lists* `<ol>`
- *Unordered lists* `<ul>`

```
<ol>
  <li>one</li>
  <li>two</li>
</ol>
<ul>
  <li>list item</li>
  <li>list item</li>
</ul>
```

`<img>` Image Element

HTML image `<img>` elements embed images in documents. The `src` attribute contains the image URL and is mandatory. `<img>` is an *empty element* meaning it should not have a closing tag.

```
<img src="image.png">
```

`alt` Attribute

An `<img>` element can have alternative text via the `alt` attribute. The alternative text will be displayed if an image fails to render due to an incorrect URL, if the image format is not supported by the browser, if the image is blocked from being displayed, or if the image has not been received from the URL.

```
<img src="path/to/image" alt="text describing image"
/>
```

HTML Attributes

HTML attributes are values added to the opening tag of an element to configure the element or change the element's default behavior. In the provided example, we are giving the `<p>` (paragraph) element a unique identifier using the `id` attribute and changing the color of the default text using the `style` attribute.

```
<p id="my-paragraph" style="color: green;">Here's
some text for a paragraph that is being altered by
HTML attributes<p>
```

Attribute Name and Values

HTML attributes consist of a name and a value using the following syntax: `name="value"` and can be added to the opening tag of an HTML element to configure or change the behavior of the element.

```
<elementName name="value"></elementName>
```

`<ol>` Ordered List Elements

HTML ordered list, `<ol>`, elements are used to create lists of items with a sequential order. Each list item in an ordered list appears numbered by default. For example:

```
<ol>
  <li>First Item</li>
  <li>Second Item</li>
  <li>Third Item</li>
</ol>
```

1. Code every day.
2. Build a portfolio.
3. Never give up.

`<title>` Title Element

The `<title>` element contains a text that defines the title of an HTML document. The title is displayed in the browser's title bar or tab in which the HTML page is displayed. The `<title>` element can only be contained inside a document's `<head>` element.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Title of the HTML page</title>
  </head>
<html>
```

`<a>` Anchor Element

The `<a>` anchor element is used to create hyperlinks in an HTML document. The hyperlinks can point to other webpages, files on the same server, a location on the same page, or any other URL via the hyperlink reference attribute, `href`. The `href` determines the location the anchor element points to.

```
<!-- Hyperlink to Codecademy -->
<a href="https://www.codecademy.com">Codecademy</a>
```

`<target>` Attribute

The `target` attribute on an anchor (`<a>`) element specifies where a hyperlink should be opened. For example, a `target` value of `"_blank"` will tell the browser to open the hyperlink in a new tab in modern browsers, or in a new window in older browsers or if the browser has had settings changed to open hyperlinks in a new window.

```
<a href="https://www.google.com" target="_blank">This anchor element links to google and will open in a new tab or window.</a>
```

`<video>` Video Element

The `<video>` element embeds a media player into an HTML document for video playback. The `src` attribute will contain the URL to the video. Adding the `controls` attribute will display video controls in the media player.

The `<video>` element can contain text content between its opening and closing tags which will be displayed if the browser does not support the video format or if the video fails to load.

```
<video src="test-video.mp4" controls>
  Video not supported
</video>
```

`<html>` Element

The `<html>` element, the root of an HTML document, should be added after the `!DOCTYPE` declaration. All content/structure for an HTML document should be contained between the opening and closing `<html>` tags.

```
<!DOCTYPE html>
<html>
  <!-- I'm a comment -->
<html>
```

File Path

URL paths in HTML can be absolute paths, like a full URL, for example: `https://developer.mozilla.org/en-US/docs/Learn` or a relative file path that links to a local file in the same folder or on the same server, for example: `./style.css`. Relative file paths begin with `./` followed by a path to the local file. `./` tells the browser to look for the file path from the current folder.

```
<a href="https://developer.mozilla.org/en-US/docs/Web">The URL for this anchor element is an absolute file path.</a>
<a href="./about.html">The URL for this anchor element is a relative file path.</a>
```

Linking to Spot on the Same Page

The HTML anchor element, `<a>` can create hyperlinks to different parts of the same HTML document using the `href` attribute to point to the desired location with `#` followed by the `id` of the element to link to.

```
<div>
  <p id="id-of-element-to-link-to">A different part of the page!</p>
</div>

<a href="#id-of-element-to-link-to">Take me to a different part of the page</a>
```

Whitespace

Whitespace, such as line breaks, added to an HTML document between block-level elements will generally be ignored by the browser and are not added to increase spacing on the rendered HTML page. Rather, whitespace is added for organization and easier reading of the HTML document itself.

```
<p>Test paragraph</p>

<!-- The whitespace created by this line, and above/below this line is ignored by the browser-->

<p>Another test paragraph, this will sit right under the first paragraph, no extra space between.</p>
```

Indenting Nested in HTML

HTML code should be formatted such that the indentation level of text increases once for each level of nesting. The W3C standards recommend two spaces of indentation per level of nesting.

```
<div>
  <h1>Heading</h1>

  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Indenting Nested in HTML

HTML code should be formatted such that the indentation level of text increases once for each level of nesting. The W3C standards recommend two spaces of indentation per level of nesting.

```
<div>
  <h1>Heading</h1>

  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Comments

In HTML, comments can be added between an opening `<!--` and closing `-->`. Content inside of comments will not be rendered by browsers, and are usually used to describe a part of code or provide other details.

Comments can span single or multiple lines.

```
<!-- Main site content -->
<div>Content</div>

<!--
  Comments can be
  multiple lines long.
-->
```

`<head>` Element

The `<head>` element contains general information about an HTML page that isn't displayed on the page itself. This information is called metadata and includes things like the title of the HTML document and links to stylesheets.

```
<!DOCTYPE html>
<html>
  <head>
    <!-- metadata is contained in this element-->
  </head>
<html>
```

Introduction to HTML : Elements and Structure

codecademy

`<table>` Table Element

In HTML, the `<table>` element has content that is used to represent a two-dimensional table made of rows and columns.

```
<table>
  <!-- rows and columns will go here -->
</table>
```

`<tr>` Table Row Element

In HTML, the table row element, `<tr>`, is used to add rows to a table before adding table data and table headings.

```
<table>
  <tr>
    ...
        </tr>
</table>
```

`<td>` Table Data Element

The table data element, `<td>`, can be nested inside a table row element, `<tr>`, to add a cell of data to a table.

```
<table>
  <tr>
        <td>cell one data</td>
        <td>cell two data</td>
        </tr>
</table>
```

`colspan` Attribute

The `colspan` attribute on a table header `<th>` or table data `<td>` element indicates how many columns that particular cell should span within the table. The `colspan` value is set to 1 by default and will take any positive integer between 1 and 1000.

```
<table>
  <tr>
    <th>row 1:</th>
    <td>col 1</td>
    <td>col 2</td>
    <td>col 3</td>
  </tr>
  <tr>
    <th>row 2:</th>
    <td colspan="2">col 1 (will span 2 columns)</td>
    <td>col 2</td>
    <td>col 3</td>
  </tr>
</table>
```

`rowspan` Attribute

Similar to `colspan`, the `rowspan` attribute on a table header or table data element indicates how many rows that particular cell should span within the table. The `rowspan` value is set to 1 by default and will take any positive integer up 65534.

```
<table>
  <tr>
    <th>row 1:</th>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
  <tr>
    <th rowspan="2">row 2 (this row will span 2
rows):</th>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
  <tr>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
   <tr>
    <th>row 3:</th>
    <td>col 1</td>
    <td>col 2</td>
  </tr>
</table>
```

`<tbody>` Table Body Element

The table body element, `<tbody>`, is a semantic element that will contain all table data other than table heading and table footer content. If used, `<tbody>` will contain all table row `<tr>` elements, and indicates that `<tr>` elements make up the body of the table. `<table>` cannot have both `<tbody>` and `<tr>` as direct children.

```
<table>
 <tbody>
   <tr>
     <td>row 1</td>
   </tr>
   <tr>
     <td>row 2</td>
   </tr>
   <tr>
     <td>row 3</td>
   </tr>
 </tbody>
</table>
```

`<thead>` Table Head Element

The table head element, `<thead>`, defines the headings of table columns encapsulated in table rows.

```
<table>
 <thead>
   <tr>
     <th>heading 1</th>
     <th>heading 2</th>
     <th>heading 3</th>
   </tr>
 </thead>
 <tbody>
   <tr>
     <td>col 1</td>
     <td>col 2</td>
     <td>col 3</td>
   </tr>
 </tbody>
</table>
```

`<tfoot>` Table Footer Element

The table footer element, `<tfoot>`, uses table rows to give footer content or to summarize content at the end of a table.

```
<table>
 <thead>
   <tr>
     <th>heading 1</th>
     <th>heading 2</th>
     <th>heading 3</th>
   </tr>
 </thead>
 <tbody>
   <tr>
     <td>col 1</td>
     <td>col 2</td>
     <td>col 3</td>
   </tr>
 </tbody>
 <tfoot>
   <tr>
     <td>summary of col 1</td>
     <td>summary of col 2</td>
     <td>summary of col 3</td>
   </tr>
 </tfoot>
</table>
```

`<th>` Table Heading Element

In HTML, the table heading element, `<th>`, is used to add titles to rows and columns of a table and must be enclosed in a table row element, `<tr>`.

```
<table>
  <tr>
        <th>column one</th>
        <th>column two</th>
        <tr>
        <td>1</td>
                <td>2</td>
        </tr>
</table>
```

Introduction to JavaScript: Introduction

codecademy

Console.log()

The `console.log()` method is used to log or print messages to the console. It can also be used to print objects and other info.

```
console.log('Hi there!');
// Prints: Hi there!
```

JavaScript Strings

Strings are a primitive data type. They are any grouping of characters (letters, spaces, numbers, or symbols) surrounded by single quotes `'` or double quotes `"`.

```
let single = 'Wheres my bandit hat?';
let double = "Wheres my bandit hat?";
```

JavaScript Variables

Variables are used whenever there's a need to store a piece of data. A variable contains data that can be used in the program elsewhere. Using variables also ensures code re-usability since it can be used to replace the same value in multiple places.

```
const incomeCurrency = '$';
let userIncome = 85000;

console.log(incomeCurrency + userIncome + ' is more
than the average income.');
// Prints: $85000 is more than the average income.
```

JavaScript Numbers

Numbers are a primitive data type. They include the set of all integers and floating point numbers.

```
let x = 2;
let y = 2.00;
```

JavaScript Booleans

Booleans are a primitive data type. They can be either `true` or `false`.

```
let lateToWork = true;
```

JavaScript Libraries

JavaScript libraries contain methods that you can call by appending the library name with a period `.`, the method name, and a set of parentheses.

```
Math.random();
// ☝ Math is the library
```

Math.random()

`Math.random()` returns a floating-point, random number in the range from 0 inclusive up to but not including 1.

```
console.log(Math.random());
// Output: 0 - 0.9
```

String.length

The `length` property of a string returns the number of characters that make up the string.

```
let x = 'good nite~';
console.log(x.length);
// Expected output: 10

console.log('howdy'.length);
// Expected output: 5
```

JavaScript Methods

Methods return information about an object, and are called by appending an instance with a period `.`, the method name, and parentheses.

```
// Returns a number between 0 and 1.
Math.random();
```

JavaScript String Concatenation

In JavaScript, multiple strings can be concatenated together using the `+` operator. In the example, multiple strings and variables containing string values have been concatenated. After execution of the code block, the `displayText` variable will contain the concatenated string.

```
let service = 'credit card';
let month = 'May 30th';
let displayText = 'Your ' + service  + ' bill is due
on ' +  month + '.';

console.log(displayText);
// output: Your credit card bill is due on May 30th.
```

Math.floor()

The `Math.floor()` function returns the largest integer less than or equal to the given number.

```
let number = 15.95;
console.log(Math.floor(number));  // 15

number = -99.5 ;
console.log(Math.floor(number));  // -100
```

Single Line Comments

In JavaScript, single-line comments are created with two consecutive forward slashes `//`.

```
let score; // This is a comment.
```

Multi-line Comments

In JavaScript, multi-line comments are created by surrounding the lines with `/*` at the beginning and `*/` at the end. Comments are good ways for a variety of reasons like explaining a code block or indicating some hints, etc.

```
/*
The below configuration must be
changed before deployment.
*/

let baseUrl = 'localhost/taxwebapp/country';
```

`const` Keyword

A constant variable can be declared using the keyword `const`. It must have an assignment. Any attempt of re-assigning a `const` variable will result in JavaScript runtime error.

```
const numberOfColumns = 4;
numberOfColumns = 8;
// TypeError: Assignment to constant variable.
```

JavaScript String Interpolation

String interpolation is the process of evaluating string literals containing one or more placeholders (expressions, variables, etc). The two main ways to interpolate strings are with:

- String concatenation: `"string" + expression + "string"`
- Template literals: `` `text ${expression} text` ``

```
let age = 7;

// String concatenation
'Tommy is ' + age + ' years old.';

// Template literal with interpolation
`Tommy is ${age} years old.`;
```

JavaScript Null

Null is a primitive data type. It represents the intentional absence of value. In code, it is represented as `null`.

```
let x = null;
```

Arithmetic Operators

JavaScript supports arithmetic operators for:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo

```
// Addition
5 + 5
// Subtraction
10 - 5
// Multiplication
5 * 10
// Division
10 / 5
// Modulo
10 % 5
```

`let` Keyword

`let` creates a local variable in JavaScript & can be re-assigned. Initialization during the declaration of a `let` variable is optional. A `let` variable will contain `undefined` if nothing is assigned to it.

```
let count;
console.log(count); // Logs undefined in console
count = 10;
console.log(count); // Logs 10 in console
```

JavaScript Undefined

`undefined` is a primitive JavaScript value that represents lack of defined value. Variables that are declared but not initialized to a value will have the value `undefined`.

```
var a;
console.log(a); // undefined
let b;
console.log(b); // undefined
```

Assignment Operators

In JavaScript, the addition assignment operator (`+=`) can be used to add the value on the right hand side to the existing value & assign it to the variable. The addition assignment operator is a shorthand for `variableName = variableName + value`. Here are all of them:

- `+=` addition assignment
- `-=` Subtraction assignment
- `*=` multiplication assignment
- `/=` division assignment

```
let number = 100;

// Both statements will add 10 to the number
number = number + 10;
number += 10;

console.log(number); // Output: 120
```

JavaScript String Interpolation

String interpolation is the process of evaluating string literals containing one or more placeholders (expressions, variables, etc). The two main ways to interpolate strings are with:

- String concatenation: `"string" + expression + "string"`
- Template literals: `` `text ${expression} text` ``

```
let age = 7;

// String concatenation
'Tommy is ' + age + ' years old.';

// Template literal with interpolation
`Tommy is ${age} years old.`;
```

JavaScript Template Literals

In JavaScript, template literals are strings that allow embedded expressions (`${expression}`). While regular strings use single (`'`) or double (`"`) quotes, template literals use backticks instead. Take a look at the code block for examples.

```
// Syntax:
`string text ${expression} more string text`

let name = "Codecademy";
console.log(`Hello, ${name}!`); // "Hello,
Codecademy!"

console.log(`Billy is ${6+8} years old.`) // "Billy
is 14 years old."
```

Introduction to JavaScript: Control Flow

codecademy

`if` Statement

An `if` statement accepts an expression with a set of parentheses `( )`:

- If the expression evaluates to a truthy value, then the code within its code body executes.
- If the expression evaluates to a falsy value, its code body will not execute.

```
const isMailSent = true;

if (isMailSent) {
  // This code block will be executed
  console.log('Mail sent to recipient');
}
```

`else` Statement

An `else` block is added to an `if` block or series of `if`-`else if` blocks. The `else` block will be executed only if the `if` condition fails.

```
const isTaskCompleted = false;

if (isTaskCompleted) {
  console.log('Task completed');
} else {
  console.log('Task incomplete');
}
```

`if` `else` `else if` Statement

After an initial `if` block, `else if` blocks can each check an additional condition. An optional `else` block can be added after the `else if` block(s) to run by default if none of the conditionals evaluated to truthy.

```
const size = 10;
if (size > 100) {
  console.log('Big!'):
} else if (size > 20) {
  console.log('Medium');
} else if (size > 4) {
  console.log('Small');
} else {
  console.log ('Tiny');
}
// 'Small'
```

`switch` Statement

JavaScript's `switch` statements provide a means of checking an expression against multiple values. It first evaluates an expression. The result of the expression is matched against values in one or more `case` clauses. If a case matches, the code inside that clause is executed.

The `case` clause should finish with a `break` keyword. If no case matches but a `default` clause is included, the code inside `default` will be executed. If `break` is omitted from the block of a `case` (or the execution is not broken otherwise, such as returning from a function with a switch), the `switch` statement will continue to check against `case` values until a break is encountered or the flow is broken.

```
const food = 'pizza';

switch (food) {
  case 'oyster':
    console.log('Enjoy the taste of the sea');
    break;
  case 'pizza':
    console.log('Enjoy a delicious pie');
    break;
  default:
    console.log('Enjoy your meal');
}

// Output: 'Enjoy a delicious pie'
// If food = 'Cheese', Output: 'Enjoy your meal'
```

JavaScript Strict Comparisons

The strict equality operator (`===`) checks if two values are the same and returns `true` if they are the same. The inequality comparison operator (`!==`) check if two values are different and return `true` if they aren't the same.

```
console.log(1 === 1);    // true
console.log('1' === 1);  // false
console.log(8 !== 9);    // true
```

JavaScript Comparison Operators

JavaScript *comparison operators* are used to compare two values and return `true` or `false` depending on the validity of the comparison. Comparison operators include:

- strict equal (`===`)
- strict not equal (`!==`)
- greater than (`>`)
- less than (`<`)
- greater than or equal (`>=`)
- less than or equal (`<=`)

```
1 > 3 // false
3 > 1 // true
250 >= 250 // true
1 === 1 // true
1 === 2 // false
1 === '1' // false
```

AND `&&` Operator

The logical AND operator `&&` checks two values and returns a boolean. If *both* values are truthy, then it returns `true`. If one, or both, of the values is falsy, then it returns `false`.

```
true && true;      // true
1 > 2 && 2 > 1;    // false
true && false;     // false
4 === 4 && 3 > 1;  // true
```

OR `||` Operator

The logical OR operator `||` checks two values and returns a boolean. If one or both values are truthy, it returns `true`. If both values are falsy, it returns `false`.

```
true || false;        // true
10 > 5 || 10 > 20;    // true
false || false;       // false
10 > 100 || 10 > 20;  // false
```

NOT `!` Operator

The `!` operator can be used to do one of the following:

- Invert a Boolean value.
- Invert the truthiness of non-Boolean values.

```
// example 1
let value = true;
let oppositeValue = !value;
console.log(oppositeValue); // false

// example 2
const emptyString = '';
!emptyString; // true
const truthyNumber = 1;
!truthyNumber // false
```

JavaScript Ternary Operator

The ternary operator allows for a compact syntax in the case of binary (choosing between two choices) decisions. It accepts a condition followed by a `?` operator, and then two expressions separated by a `;`. If the condition evaluates to truthy, the first expression is executed, otherwise, the second expression is executed. It can be read as “IF condition THEN expression1 ELSE expression2”.

```
let price = 10.5;
let day = "Monday";

// The following examples produce the same result:

// A: if/else
if (day === "Monday") {
  price -= 1.5;
} else {
  price += 1.5;
}

// B: ternary operator
day === "Monday" ? price -= 1.5 : price += 1.5;
```

Introduction to JavaScript: Functions

JavaScript Functions

Functions are one of the fundamental building blocks in JavaScript. A *function* is a reusable set of statements to perform a task or calculate a value. Functions can be *passed* one or more values and can *return* a value at the end of their execution. In order to use a function, you must define it somewhere in the scope where you wish to call it. The example code provided contains a function that takes in 2 values and returns the sum of those numbers.

```
// Defining the function:
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function:
sum(3, 6); // 9
```

JavaScript Function Declarations

Function *declarations* are used to create named functions. These functions can be called using their declared name. Function declarations are built from:

- The `function` keyword.
- The function name.
- An optional list of parameters separated by commas enclosed by a set of parentheses `()`.
- A function body enclosed in a set of curly braces `{}`.

```
function add(num1, num2) {
  return num1 + num2;
}
```

Calling JavaScript Functions

JavaScript functions can be *called*, or executed, elsewhere in code using parentheses following the function name. When a function is called, the code inside its function body runs. *Arguments* are values passed into a function when it is called.

```
// Defining the function
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function, passing in 2 and 4 as
parameters
sum(2, 4); // 6
```

JavaScript Function Parameters

Inputs to functions are known as *parameters* when a function is declared or defined. Parameters are used as variables inside the function body. When the function is called, these parameters will have the value of whatever is *passed* in as arguments. It is possible to define a function without parameters.

```
// The parameter is name
function sayHello(name) {
  return `Hello, ${name}!`;
}
```

`return` Keyword in JavaScript

Functions *return* (pass back) values using the `return` keyword. `return` ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the `return` keyword, in which case the function will return `undefined` by default.

```
// With return
function sum(num1, num2) {
  return num1 + num2;
}

// Without return, so the function doesn't output the
sum
function sum(num1, num2) {
  num1 + num2;
}
```

JavaScript Anonymous Functions

Anonymous functions in JavaScript do not have a name property. They can be defined using the `function` keyword, or as an arrow function. See the code example for the difference between a named function and an anonymous function.

```
// Named function
function rocketToMars() {
  return 'BOOM!';
}

// Anonymous function
const rocketToMars = function() {
  return 'BOOM!';
}
```

JavaScript Function Expressions

Function *expressions* create functions inside an expression instead of as a function declaration. They can be anonymous and/or assigned to a variable.

```
const dog = function() {
  return 'Woof!';
}
```

Arrow Functions in JavaScript (ES6)

Arrow function expressions were introduced in ES6. These expressions are clean and concise. The syntax for an arrow function expression does not require the `function` keyword and uses a fat arrow `=>` to separate the parameter(s) from the body. There are several variations of arrow functions:

- Arrow functions with a single parameter do not require `()` around the parameter list.
- Arrow functions with a single expression can use the concise function body which returns the result of the expression without the `return` keyword.

```
// Arrow function with two arguments
const sum = (firstParam, secondParam) => {
  return firstParam + secondParam;
};
console.log(sum(2,5)); // Prints: 7

// Arrow function with no arguments
const printHello = () => {
  console.log('hello');
};
printHello(); // Prints: hello

// Arrow functions with a single argument
const checkWeight = weight => {
  console.log(`Baggage weight : ${weight}
kilograms.`);
};
checkWeight(25); // Prints: Baggage weight : 25
kilograms.


// Concise arrow functions
const multiply = (a, b) => a * b;
console.log(multiply(2, 30)); // Prints: 60
```

Introduction to JavaScript: Scope

Scope

Scope is a concept that refers to where values and functions can accessed.

Various scopes include:

- *Global* scope (a value/function in the global scope can be used anywhere inthe entire program),
- *File* or *module* scope (the value/function can only be accessed from within the file),
- *Function* scope (only visible within the function),
- *Code block* scope (only visible within a `{ ... }` codeblock).

```
// code here can NOT use pizzaName

function myFunction() {
  var pizzaName = "Volvo";

  // code here CAN use pizzaName

}

// code here can NOT use pizzaName
```

Block Scoped Variables

`const` and `let` are *block scoped* variables, meaning they are only accessible in their block or nested blocks. In the given code block, trying to print the `statusMessage` using the `console.log()` method will result in a `ReferenceError`. It is accessible only inside that `if` block.

```
const isLoggedIn = true;

if (isLoggedIn == true) {
  const statusMessage = 'User is logged in.';
}

console.log(statusMessage);

// Uncaught ReferenceError: statusMessage is not
defined
```

Global Variables in JavaScript

JavaScript variables which are declared outside of blocks or functions can exist in the *global scope*, which means they are accessible throughout a program. Variables declared outside of smaller block or function scopes are accessible inside those smaller scopes.

It is best practice to keep global variables to a minimum, unless they must be shared across multiple blocks or functions.

```
// Variable declared globally
const color = 'blue';

function printColor() {
  console.log(color);
}

printColor(); // Prints: blue
```

Learn C++ : Hello World

codecademy

New Line

The escape sequence `\n` (backward slash and the letter n) generates a new line in a text string.

```
std::cout << "Hello\n";
std::cout << "Hello again\n";
```

Program Structure

The program runs line by line, from top to bottom:

- The first line instructs the compiler to locate the file that contains a library called `iostream`. This library contains code that allows for input and output.
- The `main()` function houses all the instructions for the program.

```
#include <iostream>

int main() {

  std::cout << "1\n";
  std::cout << "2\n";
  std::cout << "3\n";

}
```

Basic Output

`std::cout` is the “character output stream” and it is used to write to the standard output. It is followed by the symbols `<<` and the value to be displayed.

```
std::cout << "Hello World!";
```

Compile Command

In C++, the compilation command is `g++` followed by the file name.

Here, the name of the source file is **hello.cpp**.

```
g++ hello.cpp
```

Execute Command

In C++, the execution command is `./` followed by the file name.

Here, the name of the executable file is **a.out**.

```
./a.out
```

Single-line Comments

Single-line comments are created using two consecutive forward slashes. The compiler ignores any text after `//` on the same line.

```
// This line will denote a comment in C++
```

Multi-line Comments

Multi-line comments are created using `/*` to begin the comment, and `*/` to end the comment. The compiler ignores any text in between.

```
/*
This is all commented out.
None of it is going to run!
*/
```

Learn C++ : Variables

codecademy

User Input

In C++, `std::cin`, which stands for “character input”, can read user input from the keyboard.

Here, the user can enter a number, press `enter`, and that number gets stored in the variable `tip`.

```
int tip = 0;

std::cout << "Enter amount: ";
std::cin >> tip;
```

Variables

A *variable* refers to a storage location in the computer’s memory that one can set aside to save, retrieve, and manipulate data.

Variables are denoted by a name.

```
// declare a variable
int score;

// initialize a variable
score = 0;
```

Arithmetic Operators

C++ supports arithmetic operators for:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo (yields the remainder)

```
int x = 0;

x = 4 + 2;    // x is now 6
x = 4 - 2;    // x is now 2
x = 4 * 2;    // x is now 8
x = 4 / 2;    // x is now 2
x = 4 % 2;    // x is now 0
```

`int` Data Type

In C++, `int` is a type for storing integer (whole) numbers.

```
int age = 28;
```

`double` Data Type

In C++, `double` is a type for storing floating point (decimal) numbers.

```
double price = 8.99;
```

Chaining the Output

In C++, `std::cout` can output multiple values by *chaining* them using the output operator `<<`.

Here, the output would be `I'm 28.`

```
int age = 28;

std::cout << "I'm " << age << ".\n";
```

`char` Data Type

In C++, `char` is a type for storing individual characters. Characters are wrapped in single quotes.

```
char grade = 'A';
```

`string` Data Type

In C++, `std::string` is a type for storing text strings. Strings are wrapped in double quotes.

```
std::string message = "good nite";
```

`bool` Data Type

In C++, `bool` is a type for storing `true` or `false` boolean values.

```
bool late_to_work = true;
```

Learn C++ : Conditionals & Logic

codecademy

`if` Statement

In C++, an `if` statement is used to test an expression for truth.

- If the condition evaluates to `true`, then the code within the block is executed; otherwise, it will be skipped.

```
if (a == 10) {
  // code goes here
}
```

`else` Clause

In C++, an `else` clause can be added to an `if` statement.

- If the condition evaluates to `true`, code in the `if` part is executed.
- If the condition evaluates to `false`, code in the `else` part is executed.

```
if (year == 1991) {
  // executed if it is true
}
else {
  // executed if it is false
}
```

`switch` Statement

In C++, a `switch` statement is an alternative to the `if`/`else if`/`else` statement.

`switch` statement contains an expression and then various cases. The value of the expression is compared with the value of each `case`; if there is a match, the code within starts to execute.

The `break` keyword tells the computer to exit the block and not check any other cases.

`default` is executed when no case matches. It functions as the `else` clause of a `switch` statement.

```
switch (grade) {
  case 9:
    std::cout << "Freshman\n";
    break;
  case 10:
    std::cout << "Sophomore\n";
    break;
  case 11:
    std::cout << "Junior\n";
    break;
  case 12:
    std::cout << "Senior\n";
    break;
  default:
    std::cout << "Invalid\n";
    break;
}
```

Relational Operators

In C++, *relational operators* are used to compare two values:

- `==` equal to
- `!=` not equal to
- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

```
if (a > 10) {
   // ☝ means greater than
}
```

`else if` Statement

In C++, one or more `else if` statements can be added in between the `if` and `else`.

- If the `if` condition evaluates to `true`, code in the `if` part is executed.
- If the `if` condition evaluates to `false` and the `else if` condition evaluates to `true`, code in the `else if` part is executed.
- If none of the conditions evaluates to true, code in the `else` part is executed.

```
if (apple > 8) {
  // some code here
}
else if (apple > 6) {
  // some code here
}
else {
  // some code here
}
```

Logical Operators

In C++, *logical operators* can be used to combine two different conditions.

- `&&` the logical operator (`and`)
- `||` the logical operator (`or`)
- `!` the logical operator (`not`)

The `&&` requires both conditions to be `true`. The `||` requires either of the condition to be `true`. The `!` negates the result.

```
if (coffee > 0 && donut > 1) {
  // executed if both are true
}

if (coffee > 0 || donut > 1) {
  // executed if either is true
}

if (!tired) {
  // excuted if tired is false
}
```

Learn C++ : Loops

codecademy

`while` Loop

In C++, a `while` loop statement repeatedly executes the code block within as long as the condition is `true`. The moment the condition becomes `false`, the program will exit the loop.

Note that the `while` loop might not ever run. If the condition is `false` initially, the code block will be skipped.

```
while (guess != 8) {

  std::cout << "Try again: ";
  std::cin >> guess;

}
```

`for` Loop

In C++, a `for` loop executes a code block a specific number of times. It has three parts:

- The initialization of a counter
- The continue condition
- The increment/decrement of the counter

This example prints 0 to 9 on the screen.

```
for (int i = 0; i < 10; i++) {

  std::cout << i << "\n";

}
```

Learn C++ : Functions

codecademy

Functions

In C++, a *function* is a set of statements which are executed together when the function is called. Every function has a name, which is used to call the respective function.

```
#include <iostream>

// declaring a function
void print();

int main() {
  // calling a function
  print();
}

// defining a function
void print() {
  std::cout << "Hello World!";
}
```

Built-in Functions

C++ has many built-in functions. In order to use them, we have to import the required library using `#include`.

```
#include <iostream>
#include <cmath>

int main() {

  // sqrt() is from cmath
  std::cout << sqrt(10);

}
```

Calling a Function

In C++, when we define a function, it is not executed automatically. To execute it, we need to "call" the function by specifying its name followed by a pair of parentheses `()`.

```
// calling a function
print();
```

Function Declaration & Definition

A C++ function has two parts:

- Function declaration
- Function definition

The declaration includes the function's name, return type, and any parameters.

The definition is the actual body of the function which executes when a function is called. The body of a function is typically enclosed in curly braces.

```cpp
#include <iostream>

// function declaration
void blah();

// main function
int main() {
  blah();
}

// function definition
void blah() {
  std::cout << "Blah blah";
}
```

Return Values in Functions

A function that returns a value must have a `return` statement. The data type of the return value also must match the method's declared return type;

On the other hand, a `void` function (one that does not return anything) does not require a `return` statement.

```cpp
#include <iostream>

int sum(int a, int b);

int main() {

  int r = sum(10, 20);
  std::cout << r;

}

int sum(int a, int b) {

  return(a + b);

}
```

`void` Functions

In C++, if we declare the type of a function as `void`, it does not return a value. These functions are useful for a set of statements that do not require returning a value.

```cpp
#include <iostream>

void print() {
  std::cout << "Hello World!";
}

int main() {
  print();
}
```

Parameters

In C++, function parameters are placeholders for values passed to the function. They act as variables inside a function.

```
#include <iostream>

void print(int);

int main() {
  print(10);
}

// x is a parameter which holds a value of 10 when
it's called
void print(int x) {
  std::cout << x;
}
```

Function Arguments

In C++, the values passed to a function are known as arguments. They represent the actual input values.

```
#include <iostream>

void print(int);

int main() {
  print(10);
  // the argument 10 is received as input value
}

// parameter a is defined for the function print
void print(int a) {
  std::cout << a;
}
```

Scope of Code

The *scope* is the region of code that can access or view a given element:

- Variables defined in *global scope* are accessible throughout the program.
- Variables defined in a function have *local scope* and are only accessible inside the function.

```
#include <iostream>

void print();

int i = 10;       // global variable

int main() {
  std::cout << i << "\n";
}

void print() {
  int j = 0;      // local variable
  i = 20;
  std::cout << i << "\n";
  std::cout << j << "\n";
}
```

Function Declarations in Header file

C++ functions typically have two parts: declaration and definition.

Function declarations are generally stored in a *header file* (**.hpp** or **.h**) and function definitions (body of the function that defines how it is implemented) are written in the **.cpp** file.

```cpp
// ~~~~~~ main.cpp ~~~~~~

#include <iostream>
#include "functions.hpp"

int main() {

  std::cout << say_hi("Sabaa");

}


// ~~~~~~ functions.hpp ~~~~~~

// function declaration
std::string say_hi(std::string name);


// ~~~~~~ functions.cpp ~~~~~~

#include <string>
#include "functions.hpp"

// function defintion
std::string say_hi(std::string name) {

  return "Hey there, " + name + "!\n";

}
```

Learn C++ : Classes and Objects

codecademy

Destructor

For a C++ class, a *destructor* is a special method that handles object destruction, generally focused on preventing memory leaks. Class destructors don't take arguments as input and their name is always preceded by a tilde `~`.

```
City::~City() {

  // any final cleanup

}
```

Class Members

A C++ class is comprised of class members:

- *Attributes*, also known as member data, consist of information about an instance of the class.
- *Methods*, also known as member functions, are functions that can be used with an instance of the class.

```
class City {

  // attribute
  int population;

public:
  // method
  void add_resident() {
    population++;
  }

};
```

Constructor

For a C++ class, a *constructor* is a special kind of method that enables control regarding how the objects of a class should be created. Different class constructors can be specified for the same class, but each constructor signature must be unique.

```
#include "city.hpp"

class City {

  std::string name;
  int population;

public:
  City(std::string new_name, int new_pop);

};
```

Objects

In C++, an *object* is an instance of a class that encapsulates data and functionality pertaining to that data.

```
City nyc;
```

Access Control Operators

C++ classes have access control operators that designate the scope of class members:

- `public`
- `private`

`public` members are accessible everywhere; `private` members can only be accessed from within the same instance of the class or from friends classes.

```
class City {

  int population;

public:
  void add_resident() {
    population++;
  }

private:
  bool is_capital;
};
```

Learn C++ : References and Pointers

codecademy

`const` Reference

In C++, pass-by-reference with `const` can be used for a function where the parameter(s) won't change inside the function.

This saves the computational cost of making a copy of the argument.

```
int triple(int const &i) {

  return i * 3;

}
```

Pointers

In C++, a *pointer* variable stores the memory address of something else. It is created using the `*` sign.

```
int* ptr = &gum;
```

References

In C++, a *reference* variable is an alias for another object. It is created using the `&` sign. Two things to note:

1. Anything done to the reference also happens to the original.
2. Aliases cannot be changed to alias something else.

```
int &sonny = songqiao;
```

Memory Address

In C++, the *memory address* is the location in the memory of an object. It can be accessed with the "address of" operator, `&`.

Given a variable `porcupine_count`, the memory address can be retrieved by printing out `&porcupine_count`. It will return something like: `0x7ffd7caa5b54`.

```
std::cout << &porcupine_count << "\n";
```

Pass-By-Reference

In C++, *pass-by-reference* refers to passing parameters to a function by using references.

It allows the ability to:

- Modify the value of the function arguments.
- Avoid making copies of a variable/object for performance reasons.

```
void swap_num(int &i, int &j) {

  int temp = i;
  i = j;
  j = temp;

}

int main() {

  int a = 100;
  int b = 200;

  swap_num(a, b);

  std::cout << "A is " << a << "\n";
  std::cout << "B is " << b << "\n";

}
```

Learn SQL : Manipulation

codecademy

`CREATE TABLE` Statement

The `CREATE TABLE` statement is used to create new tables in a database. Column names, types and constraints are provided as a comma-separated list of values between a set of parentheses `()`:

- The column parameters specify the column names of the table.
- The data type parameters specify the type of data the column can hold (e.g. `TEXT`, `INTEGER`).
- The constraints specify the rules that apply to the values of a column.

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  column3 datatype
);
```

`INSERT` Statement

The `INSERT INTO` statement is used to add new records (rows) to a table.

It has two forms as shown in the code block: either define the columns to insert values into or insert them directly based on the order of the columns in the table.

```
-- Insert into columns in order:
INSERT INTO table_name
VALUES (value1, value2, value3);

-- Insert into columns by name:
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

`ALTER TABLE` Statement

The `ALTER TABLE` statement is used to modify the columns of an existing table. When combined with the `ADD COLUMN` clause, it is used to add a new column to a table.

```
-- Syntax:
ALTER TABLE table_name
ADD column_name datatype;

-- Example:
ALTER TABLE employees
ADD first_name TEXT;
```

`UPDATE` Statement

The `UPDATE` statement is used to edit records (rows) in a table.

The `UPDATE` statement usually includes a `SET` clause that indicates the column to edit and a `WHERE` clause for specifying which record(s) should be updated.

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE some_column = some_value;
```

Column Constraints

SQL column constraints are the rules applied to the values of individual columns:

- `PRIMARY KEY` column can be used to uniquely identify the row.
- `UNIQUE` columns have a different value for every row.
- `NOT NULL` columns must have a value; they cannot be `NULL`.
- `DEFAULT` assigns a default value for the column when no value is specified.

There can be only one `PRIMARY KEY` column per table and multiple `UNIQUE` columns.

```sql
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  name TEXT UNIQUE,
  grade INTEGER NOT NULL,
  age INTEGER DEFAULT 10
);
```

Learn SQL : Queries

codecademy

`SELECT` Statement in SQL

The `SELECT *` statement returns all columns from the provided table(s) in the result set. The given query will select all columns and records (rows) from the `movies` table.

```
SELECT *
FROM movies;
```

`AS` Clause in SQL

Columns or tables in SQL can be *aliased* using the `AS` clause. This allows columns or tables to be specifically renamed in the returned result set. The given query will return a result set with the column for `name` renamed to `movie_title`.

```
SELECT name AS 'movie_title'
FROM movies;
```

`DISTINCT` Query in SQL

Unique values of a column can be selected using a `DISTINCT` query. For a table `contact_details` having five rows in which the `city` column contains Chicago, Madison, Boston, Madison, and Denver, the given query would return:

```
SELECT DISTINCT city
FROM contact_details;
```

- `Chicago`
- `Madison`
- `Boston`
- `Denver`

`WHERE` Clause in SQL

The `WHERE` clause is used to filter records (rows) that match a certain condition. The given query will select all records where the `pub_year` equals `2017`.

```
SELECT title
FROM library
WHERE pub_year = 2017;
```

`LIMIT` Clause in SQL

The `LIMIT` clause is used to narrow, or *limit*, a result set to the specified number of rows. The given query will limit the result set to 5 rows.

```
SELECT *
FROM movies
LIMIT 5;
```

`LIKE` Operator in SQL

The `LIKE` operator can be used inside of a `WHERE` clause to match a specified pattern. The given query will match any movie that begins with `Star` in its title.

```
SELECT name
FROM movies
WHERE name LIKE 'Star%';
```

`_` Wildcard in SQL

The `_` wildcard can be used in a `LIKE` operator pattern to match any single unspecified character. The given query will match any movie which begins with a single character, followed by `ove`.

```
SELECT name
FROM movies
WHERE name LIKE '_ove';
```

`BETWEEN` Operator in SQL

The `BETWEEN` operator can be used to filter by a *range* of values. The range of values can be text, numbers or date data. The given query will match any movie made between the years 1980 and 1990, inclusive.

```
SELECT *
FROM movies
WHERE year BETWEEN 1980 AND 1990;
```

`AND` Operator in SQL

The `AND` operator allows multiple conditions to be combined. Records must match both conditions that are joined by `AND` to be included in the result set. The example query will match any car that is blue and made after 2014.

```
SELECT model
FROM cars
WHERE color = 'blue'
  AND year > 2014;
```

`OR` Operator in SQL

The `OR` operator allows multiple conditions to be combined. Records matching either condition joined by the `OR` are included in the result set. The given query will match customers whose state is either `ca` or `ny`.

```
SELECT name
FROM customers
WHERE state = "ca"
   OR state = "ny";
```

`ORDER BY` Clause in SQL

The `ORDER BY` clause can be used to sort the result set of a query by one or more columns. Using the `ORDER BY` clause, data can be ordered in ascending (default) or descending order by the `ASC` and `DESC` keywords. In the example, all the rows of the `contacts` table will be ordered by the `birth_date` column in descending order.

```
SELECT *
FROM contacts
ORDER BY birth_date DESC;
```

`%` Wildcard in SQL

The `%` wildcard can be used in a `LIKE` operator pattern to match zero or more unspecified character(s). The example query will match any movie that begins with `The`, followed by zero or more of any characters.

```
SELECT name
FROM movies
WHERE name LIKE 'The%';
```

`NULL` Column Values in SQL

Column values in SQL records can be `NULL`, or have no value. These records can be matched (or not matched) using the `IS NULL` and `IS NOT NULL` operators in combination with the `WHERE` clause. The given query will match all addresses where the address has a value or is not `NULL`.

```
SELECT address
FROM records
WHERE address IS NOT NULL;
```

Learn SQL : Aggregate Functions

codecademy

Column References

The `GROUP BY` and `ORDER BY` clauses can reference the selected columns by number in which they appear in the `SELECT` statement. The example query will count the number of movies per rating, and will:

- `GROUP BY` column `2` (`rating`)
- `ORDER BY` column `1` (`total_movies`)

```
SELECT COUNT(*) AS 'total_movies',
   rating
FROM movies
GROUP BY 2
ORDER BY 1;
```

`MAX()` Aggregate Function

The `MAX()` aggregate function in SQL takes the name of a column as an argument and returns the largest value in a column. The given query will return the largest value from the `amount` column.

```
SELECT MAX(amount)
FROM transactions;
```

`SUM()` Aggregate Function

The `SUM()` aggregate function in SQL that takes the name of a column as an argument and returns the sum of all the value in that column.

```
SELECT SUM(salary)
FROM salary_disbursement;
```

`COUNT()` Aggregate Function

The `COUNT()` aggregate function in SQL returns the total number of rows that match the specified criteria. For instance, to find the total number of employees who have more than 5 years of experience, the given query can be used.

Note: A column name of the table can also be used instead of `*`. Unlike `COUNT(*)`, this variation `COUNT(column)` will *not* count `NULL` values in that column.

```
SELECT COUNT(*)
FROM employees
WHERE experience < 5;
```

`GROUP BY` Clause

The `GROUP BY` clause will group records in a result set by identical values in one or more columns. It is often used in combination with aggregate functions to query information of similar records. The `GROUP BY` clause can come after `FROM` or `WHERE` but must come before any `ORDER BY` or `LIMIT` clause.

The given query will count the number of movies per rating.

```
SELECT rating,
   COUNT(*)
FROM movies
GROUP BY rating;
```

`MIN()` Aggregate Function

The `MIN()` aggregate function in SQL returns the smallest value in a column. For instance, to find the smallest value of the `amount` column from the table named `transactions`, the given query can be used.

```
SELECT MIN(amount)
FROM transactions;
```

`AVG()` Aggregate Function

The `AVG()` aggregate function in SQL returns the average value in a column. For instance, to find the average `salary` for the employees who have less than 5 years of experience, the given query can be used.

```
SELECT AVG(salary)
FROM employees
WHERE experience < 5;
```

`HAVING` Clause

The `HAVING` clause is used to further filter the result set groups provided by the `GROUP BY` clause. `HAVING` is often used with aggregate functions to filter the result set groups based on an aggregate property. The given query will select only the records (rows) from only years where more than 5 movies were released per year.

```
SELECT year,
   COUNT(*)
FROM movies
GROUP BY year
HAVING COUNT(*) > 5;
```

`ROUND()` Function

The `ROUND()` function will round a number value to a specified number of places. It takes two arguments: a number, and a number of decimal places. It can be combined with other aggregate functions, as shown in the given query. This query will calculate the average rating of movies from 2015, rounding to 2 decimal places.

```
SELECT year,
   ROUND(AVG(rating), 2)
FROM movies
WHERE year = 2015;
```

Learn SQL : Multiple Tables

codecademy

Inner Join

The `JOIN` clause allows for the return of results from more than one table by joining them together with other results based on common column values specified using an `ON` clause. `INNER JOIN` is the default `JOIN` and it will only return results matching the condition specified by `ON`.

In this example, results from the `books` table are joined with results from the `authors` table in order to display an author for each book through the shared value of `author_id` in both tables.

```
SELECT *
FROM books
JOIN authors
  ON books.author_id = authors.id;
```

Outer Join

The `LEFT JOIN` clause is used to combine data from tables on a common property that the user specifies using an `ON` clause. `LEFT JOIN` takes all the records (rows) from the left table and combines them with only the matching records from the right table based on the column specified in the `ON` clause. If there is no match, the corresponding right table value(s) will be set to `NULL` in the result.

In the given query, all records from `table1` are included in the result set and are combined with the records from `table2` on a record by record basis. If a record from `table1` has the same value in column `c2` that a record from `table2` has in column `c3`, the records are combined.

```
SELECT *
FROM table1
LEFT JOIN table2
  ON table1.c2 = table2.c3;
```

`WITH` Clause

The `WITH` clause stores the result of a query in a temporary table (`temporary_movies`) using an alias.

Multiple temporary tables can be defined with one instance of the `WITH` keyword.

```
WITH temporary_movies AS (
   SELECT *
   FROM movies
)
SELECT *
FROM temporary_movies
WHERE year BETWEEN 2000 AND 2020;
```

`UNION` Clause

The `UNION` clause is used to combine results that appear from multiple `SELECT` statements and filter duplicates.

For example, given a `first_names` table with a column `name` containing rows of data “James” and “Hermione”, and a `last_names` table with a column `name` containing rows of data “James”, “Hermione” and “Cassidy”, the result of this query would contain three `name`s: “Cassidy”, “James”, and “Hermione”.

```
SELECT name
FROM first_names
UNION
SELECT name
FROM last_names
```

`CROSS JOIN` Clause

The `CROSS JOIN` clause is used to combine each row from one table with each row from another in the result set. This `JOIN` is helpful for creating all possible combinations for the records (rows) in two tables.

The given query will select the `shirt_color` and `pants_color` columns from the result set, which will contain all combinations of combining the rows in the `shirts` and `pants` tables. If there are 3 different shirt colors in the `shirts` table and 5 different pants colors in the `pants` table then the result set will contain 3 x 5 = 15 rows.

```
SELECT shirts.shirt_color,
   pants.pants_color
FROM shirts
CROSS JOIN pants;
```

Learn SQL : Manipulation

codecademy

`CREATE TABLE` Statement

The `CREATE TABLE` statement is used to create new tables in a database. Column names, types and constraints are provided as a comma-separated list of values between a set of parentheses `()`:

- The column parameters specify the column names of the table.
- The data type parameters specify the type of data the column can hold (e.g. `TEXT`, `INTEGER`).
- The constraints specify the rules that apply to the values of a column.

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  column3 datatype
);
```

`INSERT` Statement

The `INSERT INTO` statement is used to add new records (rows) to a table.

It has two forms as shown in the code block: either define the columns to insert values into or insert them directly based on the order of the columns in the table.

```
-- Insert into columns in order:
INSERT INTO table_name
VALUES (value1, value2, value3);

-- Insert into columns by name:
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

`ALTER TABLE` Statement

The `ALTER TABLE` statement is used to modify the columns of an existing table. When combined with the `ADD COLUMN` clause, it is used to add a new column to a table.

```
-- Syntax:
ALTER TABLE table_name
ADD column_name datatype;

-- Example:
ALTER TABLE employees
ADD first_name TEXT;
```

`UPDATE` Statement

The `UPDATE` statement is used to edit records (rows) in a table.

The `UPDATE` statement usually includes a `SET` clause that indicates the column to edit and a `WHERE` clause for specifying which record(s) should be updated.

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE some_column = some_value;
```

Column Constraints

SQL column constraints are the rules applied to the values of individual columns:

- `PRIMARY KEY` column can be used to uniquely identify the row.
- `UNIQUE` columns have a different value for every row.
- `NOT NULL` columns must have a value; they cannot be `NULL`.
- `DEFAULT` assigns a default value for the column when no value is specified.

There can be only one `PRIMARY KEY` column per table and multiple `UNIQUE` columns.

```sql
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  name TEXT UNIQUE,
  grade INTEGER NOT NULL,
  age INTEGER DEFAULT 10
);
```

Learn Python 3 : Functions

codecademy

Python Functions

Some tasks need to be performed multiple times within a program. Rather than rewrite the same code in multiple places, a function may be defined using the `def` keyword. Function definitions may include parameters, providing data input to the function.

Functions may return a value using the `return` keyword followed by the value to return.

In the example, a function `my_function` is defined with the parameter `x`. The function returns a value that takes the parameter and adds 1 to it. It is then invoked multiple times with different input values.

```
# define a function my_function() with parameter x

def my_function(x):
  return x + 1

# invoke our function

print(my_function(2))       # outputs: 3
print(my_function(3 + 5))   # outputs: 9
```

Calling Functions

Python uses simple syntax to use, invoke, or *call* a preexisting function. A function can be called by writing the name of it, followed by parentheses.

For example, the code provided would call the `doHomework()` method.

```
doHomework()
```

Defining Functions

A developer can create or *define* his or her own function in Python. To do so, the keyword `def` is followed by the name of the function, parentheses, and a colon. The body of the function, or the code for what the function will actually do, comes after the colon on indented lines.

```
def doHomework():
  # function body goes here
```

Multiple Parameters

Python functions can have multiple *parameters*. Just as you wouldn't go to school without both a backpack and a pencil case, functions may also need more than one input to carry out their operations.

To define a function with multiple parameters, parameter names are placed one after another, separated by commas, within the parentheses of the function definition.

```
def ready_for_school(backpack, pencil_case):
  if (backpack == 'full' and pencil_case == 'full'):
    print ("I'm ready for school!")
```

Returning Multiple Values

Python functions are able to return multiple values using one `return` statement. All values that should be returned are listed after the `return` keyword and are separated by commas.

In the example, the function `square_point()` returns `x_squared`, `y_squared`, and `z_squared`.

```
def square_point(x, y, z):
  x_squared = x * x
  y_squared = y * y
  z_squared = z * z
  # Return all three values:
  return x_squared, y_squared, z_squared

three_squared, four_squared, five_squared =
square_point(3, 4, 5)
```

Returning Value from Function

A `return` keyword is used to return a value from a Python function. The value returned from a function can be assigned to a variable which can then be used in the program.

In the example, the function `check_leap_year` returns a string which indicates if the passed parameter is a leap year or not.

```
def check_leap_year(year):
  if year % 4 == 0:
    return str(year) + " is a leap year."
  else:
    return str(year) + " is not a leap year."

year_to_check = 2018
returned_value = check_leap_year(year_to_check)
print(returned_value) # 2018 is not a leap year.
```

Function Indentation

Python uses indentation to identify blocks of code. Code within the same block should be indented at the same level. A Python function is one type of code block. All code under a function declaration should be indented to identify it as part of the function. There can be additional indentation within a function to handle other statements such as `for` and `if` so long as the lines are not indented less than the first line of the function code.

```
# Indentation is used to identify code blocks

def testfunction(number):
  # This code is part of testfunction
  print("Inside the testfunction")
  sum = 0
  for x in range(number):
    # More indentation because 'for' has a code block
    # but still part of he function
    sum += x
  return sum
print("This is not part of testfunction")
```

Function Parameters

Sometimes functions require input to provide data for their code. This input is defined using *parameters*.

Parameters are variables that are defined in the function definition. They are assigned the values which were passed as arguments when the function was called, elsewhere in the code.

For example, the function definition defines parameters for a character, a setting, and a skill, which are used as inputs to write the first sentence of a book.

```
def write_a_book(character, setting, special_skill):
  print(character + " is in " +
        setting + " practicing her " +
        special_skill)
```

The Scope of Variables

In Python, a variable defined inside a function is called a local variable. It cannot be used outside of the scope of the function, and attempting to do so without defining the variable outside of the function will cause an error.

In the example, the variable `a` is defined both inside and outside of the function. When the function `f1()` is implemented, `a` is printed as `2` because it is locally defined to be so. However, when printing `a` outside of the function, `a` is printed as `5` because it is implemented outside of the scope of the function.

```
a = 5

def f1():
  a = 2
  print(a)

print(a)   # Will print 5
f1()       # Will print 2
```

Function Arguments

Parameters in python are variables — placeholders for the actual values the function needs. When the function is *called*, these values are passed in as *arguments*.

For example, the arguments passed into the function `.sales()` are the “The Farmer’s Market”, “toothpaste”, and “$1” which correspond to the parameters `grocery_store`, `item_on_sale`, and `cost`.

```
def sales(grocery_store, item_on_sale, cost):
  print(grocery_store + " is selling " + item_on_sale
+ " for " + cost)

sales("The Farmer’s Market", "toothpaste", "$1")
```

Global Variables in Python

A variable that is defined outside of a function is called a global variable. It can be accessed inside the body of a function.

In the example, the variable `a` is a global variable because it is defined outside of the function `prints_a`. It is therefore accessible to `prints_a`, which will print the value of `a`.

```
a = "Hello"

def prints_a():
  print(a)

# will print "Hello"
prints_a()
```

Function Keyword Arguments

Python functions can be defined with named arguments which may have default values provided. When function arguments are passed using their names, they are referred to as keyword arguments. The use of keyword arguments when calling a function allows the arguments to be passed in any order — *not* just the order that they were defined in the function. If the function is invoked without a value for a specific argument, the default value will be used.

```
def findvolume(length=1, width=1, depth=1):
  print("Length = " + str(length))
  print("Width = " + str(width))
  print("Depth = " + str(depth))
  return length * width * depth;

findvolume(1, 2, 3)
findvolume(length=5, depth=2, width=4)
findvolume(2, depth=3, width=4)
```

Parameters as Local Variables

Function parameters behave identically to a function's local variables. They are initialized with the values passed into the function when it was called.

Like local variables, parameters cannot be referenced from outside the scope of the function.

In the example, the parameter `value` is defined as part of the definition of `my_function`, and therefore can only be accessed within `my_function`. Attempting to print the contents of `value` from outside the function causes an error.

```
def my_function(value):
  print(value)

# Pass the value 7 into the function
my_function(7)

# Causes an error as `value` no longer exists
print(value)
```

Learn Python 3 : Control Flow

codecademy

`if` Statement

The Python `if` statement is used to determine the execution of code based on the evaluation of a Boolean expression.

- If the `if` statement expression evaluates to `True`, then the indented code following the statement is executed.
- If the expression evaluates to `False` then the indented code following the `if` statement is skipped and the program executes the next line of code which is indented at the same level as the `if` statement.

```
# if Statement

test_value = 100

if test_value > 1:
  # Expression evaluates to True
  print("This code is executed!")

if test_value > 1000:
  # Expression evaluates to False
  print("This code is NOT executed!")

print("Program continues at this point.")
```

`elif` Statement

The Python `elif` statement allows for continued checks to be performed after an initial `if` statement. An `elif` statement differs from the `else` statement because another expression is provided to be checked, just as with the initial `if` statement.

If the expression is `True`, the indented code following the `elif` is executed. If the expression evaluates to `False`, the code can continue to an optional `else` statement. Multiple `elif` statements can be used following an initial `if` to perform a series of checks. Once an `elif` expression evaluates to `True`, no further `elif` statements are executed.

```
# elif Statement

pet_type = "fish"

if pet_type == "dog":
  print("You have a dog.")
elif pet_type == "cat":
  print("You have a cat.")
elif pet_type == "fish":
  # this is performed
  print("You have a fish")
else:
  print("Not sure!")
```

`else` Statement

The Python `else` statement provides alternate code to execute if the expression in an `if` statement evaluates to `False`.

The indented code for the `if` statement is executed if the expression evaluates to `True`. The indented code immediately following the `else` is executed only if the expression evaluates to `False`. To mark the end of the `else` block, the code must be unindented to the same level as the starting `if` line.

```
# else Statement

test_value = 50

if test_value < 1:
  print("Value is < 1")
else:
  print("Value is >= 1")

test_string = "VALID"

if test_string == "NOT_VALID":
  print("String equals NOT_VALID")
else:
  print("String equals something else!")
```

Comparison Operators

In Python, *relational operators* compare two values or expressions. The most common ones are:

- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal too

If the relation is sound, then the entire expression will evaluate to `True`. If not, the expression evaluates to `False`.

```
a = 2
b = 3
a < b  # evaluates to True
a > b  # evaluates to False
a >= b # evaluates to False
a <= b # evaluates to True
a <= a # evaluates to True
```

Equal Operator `==`

The equal operator, `==`, is used to compare two values, variables or expressions to determine if they are the same.

If the values being compared are the same, the operator returns `True`, otherwise it returns `False`.

The operator takes the data type into account when making the comparison, so a string value of `"2"` is *not* considered the same as a numeric value of `2`.

```
# Equal operator

if 'Yes' == 'Yes':
  # evaluates to True
  print('They are equal')

if (2 > 1) == (5 < 10):
  # evaluates to True
  print('Both expressions give the same result')

c = '2'
d = 2

if c == d:
  print('They are equal')
else:
  print('They are not equal')
```

Not Equals Operator `!=`

The Python not equals operator, `!=`, is used to compare two values, variables or expressions to determine if they are NOT the same. If they are NOT the same, the operator returns `True`. If they are the same, then it returns `False`.

The operator takes the data type into account when making the comparison so a value of `10` would NOT be equal to the string value `"10"` and the operator would return `True`. If expressions are used, then they are evaluated to a value of `True` or `False` before the comparison is made by the operator.

```
# Not Equals Operator

if "Yes" != "No":
  # evaluates to True
  print("They are NOT equal")

val1 = 10
val2 = 20

if val1 != val2:
  print("They are NOT equal")

if (10 > 1) != (10 > 1000):
  # True != False
  print("They are NOT equal")
```

Boolean Values

Booleans are a data type in Python, much like integers, floats, and strings. However, booleans only have two values:

- `True`
- `False`

Specifically, these two values are of the `bool` type. Since booleans are a data type, creating a variable that holds a boolean value is the same as with other data types.

```
is_true = True
is_false = False

print(type(is_true))
# will output: <class 'bool'>
```

`and` Operator

The Python `and` operator performs a Boolean comparison between two Boolean values, variables, or expressions. If both sides of the operator evaluate to `True` then the `and` operator returns `True`. If either side (or both sides) evaluates to `False`, then the `and` operator returns `False`. A non-Boolean value (or variable that stores a value) will always evaluate to `True` when used with the `and` operator.

```
True and True      # Evaluates to True
True and False     # Evaluates to False
False and False    # Evaluates to False
1 == 1 and 1 < 2   # Evaluates to True
1 < 2 and 3 < 1    # Evaluates to False
"Yes" and 100      # Evaluates to True
```

`or` Operator

The Python `or` operator combines two Boolean expressions and evaluates to `True` if at least one of the expressions returns `True`. Otherwise, if both expressions are `False`, then the entire expression evaluates to `False`.

```
True or True       # Evaluates to True
True or False      # Evaluates to True
False or False     # Evaluates to False
1 < 2 or 3 < 1     # Evaluates to True
3 < 1 or 1 > 6     # Evaluates to False
1 == 1 or 1 < 2    # Evaluates to True
```

`not` Operator

The Python Boolean `not` operator is used in a Boolean expression in order to evaluate the expression to its inverse value. If the original expression was `True`, including the `not` operator would make the expression `False`, and vice versa.

```
not True      # Evaluates to False
not False     # Evaluates to True
1 > 2         # Evaluates to False
not 1 > 2     # Evaluates to True
1 == 1        # Evaluates to True
not 1 == 1    # Evaluates to False
```

Handling Exceptions in Python

A `try` and `except` block can be used to handle error in code block. Code which may raise an error can be written in the `try` block. During execution, if that code block raises an error, the rest of the `try` block will cease executing and the `except` code block will execute.

```
def check_leap_year(year):
  is_leap_year = False
  if year % 4 == 0:
    is_leap_year = True

try:
  check_leap_year(2018)
  print(is_leap_year)
  # The variable is_leap_year is declared inside the
function
except:
  print('Your code raised an error!')
```

Learn Python 3 : Lists

codecademy

Python Lists

In Python, lists are ordered collections of items that allow for easy use of a set of data.

List values are placed in between square brackets `[ ]`, separated by commas. It is good practice to put a space in between the comma and the next value. The values in a list do not need to be unique (the same value can be repeated).

Empty lists do not contain any values within the square brackets.

```
primes = [2, 3, 5, 7, 11]
print(primes)

empty_list = []
```

Python Lists: Data Types

In Python, lists are a versatile data type that can contain multiple different data types within the same square brackets. The possible data types within a list include numbers, strings, other objects, and even other lists.

```
numbers = [1, 2, 3, 4, 10]
names = ['Jenny', 'Sam', 'Alexis']
mixed = ['Jenny', 1, 2]
list_of_lists = [['a', 1], ['b', 2]]
```

Adding Lists Together

In Python, lists can be added to each other using the plus symbol `+`. As shown in the code block, this will result in a new list containing the same items in the same order with the first list's items coming first.

Note: This will not work for adding one item at a time (use `.append()` method). In order to add one item, create a new list with a single value and then use to plus symbol to add the list.

```
items = ['cake', 'cookie', 'bread']
total_items = items + ['biscuit', 'tart']
print(total_items)
# Result: ['cake', 'cookie', 'bread', 'biscuit',
'tart']
```

List Indices

Python list elements are ordered by *index*, a number referring to their placement in the list. List indices start at 0 and increment by one.

To access a list element by index, square bracket notation is used: `list[index]`.

```
berries = ["blueberry", "cranberry", "raspberry"]

berries[0]   # "blueberry"
berries[2]   # "raspberry"
```

Aggregating Iterables Using `zip()`

In Python, data types that can be iterated (called iterables) can be used with the `zip()` function to aggregate data based on the iterables passed in.

As shown in the example, `zip()` is aggregating the data between the owners' names and the dogs' names to match the owner to their dogs. `zip()` returns an iterator containing the data based on what the user passes through and can be printed to visually represent the aggregated data. Empty iterables passed in will result in an empty iterator.

```
owners_names = ['Jenny', 'Sam', 'Alexis']
dogs_names = ['Elphonse', 'Dr. Doggy DDS', 'Carter']
owners_dogs = zip(owners_names, dogs_names)
print(owners_dogs)
# Result: [('Jenny', 'Elphonse'), ('Sam', 'Dr.Doggy
DDS'), ('Alexis', 'Carter')]
```

List Item Ranges Including First or Last Item

In Python, when selecting a range of list items, if the first item to be selected is at index `0`, no index needs to be specified before the `:`. Similarly, if the last item selected is the last item in the list, no index needs to be specified after the `:`.

```
items = [1, 2, 3, 4, 5, 6]

# All items from index `0` to `3`
print(items[:4])

# All items from index `2` to the last item, inclusive
print(items[2:])
```

List Method `.count()`

The `.count()` Python list method searches a list for whatever search term it receives as an argument, then returns the number of matching entries found.

```
backpack = ['pencil', 'pen', 'notebook', 'textbook',
'pen', 'highlighter', 'pen']
numPen = backpack.count('pen')
print(numPen)
# Output: 3
```

List Method `.append()`

In Python, you can add values to the end of a list using the `.append()` method. This will place the object passed in as a new element at the very end of the list. Printing the list afterwards will visually show the appended value. This `.append()` method is *not* to be confused with returning an entirely new list with the passed object.

```
orders = ['daisies', 'periwinkle']
orders.append('tulips')
print(orders)
# Result: ['daisies', 'periwinkle', 'tulips']
```

List Method `.sort()`

The `.sort()` Python list method will sort the contents of whatever list it is called on. Numerical lists will be sorted in ascending order, and lists of Strings will be sorted into alphabetical order. It modifies the original list, and has no return value.

```
exampleList = [4, 2, 1, 3]
exampleList.sort()
print(exampleList)
# Output: [1, 2, 3, 4]
```

Determining List Length with `len()`

The Python `len()` function can be used to determine the number of items found in the list it accepts as an argument.

```
knapsack = [2, 4, 3, 7, 10]
size = len(knapsack)
print(size)
# Output: 5
```

Zero-Indexing

In Python, list index begins at zero and ends at the length of the list minus one. For example, in this list, `'Andy'` is found at index `2`.

```
names = ['Roger', 'Rafael', 'Andy', 'Novak']
```

List Item Ranges Including First or Last Item

In Python, when selecting a range of list items, if the first item to be selected is at index `0`, no index needs to be specified before the `:`. Similarly, if the last item selected is the last item in the list, no index needs to be specified after the `:`.

```
items = [1, 2, 3, 4, 5, 6]

# All items from index `0` to `3`
print(items[:4])

# All items from index `2` to the last item, inclusive
print(items[2:])
```

`sorted()` Function

The Python `sorted()` function accepts a list as an argument, and will return a new, sorted list containing the same elements as the original. Numerical lists will be sorted in ascending order, and lists of Strings will be sorted into alphabetical order. It does not modify the original, unsorted list.

```
unsortedList = [4, 2, 1, 3]
sortedList = sorted(unsortedList)
print(sortedList)
# Output: [1, 2, 3, 4]
```

Negative List Indices

Negative indices for lists in Python can be used to reference elements in relation to the end of a list. This can be used to access single list elements or as part of defining a list range. For instance:

- To select the last element, `my_list[-1]`.
- To select the last three elements, `my_list[-3:]`.
- To select everything except the last two elements, `my_list[:-2]`.

```
soups = ['minestrone', 'lentil', 'pho', 'laksa']
soups[-1]  # 'laksa'
soups[-3:] # 'lentil', 'pho', 'laksa'
soups[-2:] # 'minestrone', 'lentil'
```

List Slicing

A *slice*, or sub-list of Python list elements can be selected from a list using a colon-separated starting and ending point.

The syntax pattern is `myList[START_NUMBER:END_NUMBER]`. The slice will include the `START_NUMBER` index, and everything until but excluding the `END_NUMBER` item.

When slicing a list, a new list is returned, so if the slice is saved and then altered, the original list remains the same.

```
tools = ['pen', 'hammer', 'lever']
tools_slice = tools[1:3] # ['hammer', 'lever']
tools_slice[0] = 'nail'

# Original list is unaltered:
print(tools) # ['pen', 'hammer', 'lever']
```