Tiny ES6 Notebook

Curated JavaScript Examples

Matt Harrison

Tiny Notebook Series

Tiny ES6 Notebook

Curated JavaScript Examples

Matt Harrison

Table of Contents

Introduction
Strict Mode
Variables
Types
Numbers
Built-in Types
Built-in Functions
Unicode
Functions
Classes
Operators
Conditionals
Looping
Exceptions
Generators
Modules
Promises
Regular Expressions
About the Author
Also Available
One more thing

Introduction

THIS IS NOT SO MUCH AN INSTRUCTIONAL MANUAL, BUT RATHER NOTES, TABLES, AND EXAMPLES FOR ECMAScript2015 or ES6 syntax (aka modern JavaScript). It was created by the author as an additional resource during training, meant to be distributed as a physical notebook. Participants (who favor the physical characteristics of dead tree material) could add their own notes, thoughts, and have a valuable reference of curated examples.

Strict Mode

ES6 modules are always strict according to the spec. There should be no need to place this code at the top of your code to enable strict mode:

```
'use strict';
```

Strict mode does the following:

- Requires explicit creation of global variables (via `let`)
- Throws exceptions when assigning to a non-writable variable
- Throws errors when deleting undeletable properties
- Throws `SyntaxError` if duplicating function parameter names
- Throws a `SyntaxError` when putting a `0` in front of a numeric literal (use `0o` for an octal)
- Throws a `TypeError` when setting a property on a primitive.

Variables

There are various ways to declare a variable in ES6. Function local variables are declared with `var`, block variables are declared with `let`, and constant variables are declared with `const`:

```
const PI = 3.14

//PI = 3.1459 // TypeError

function add(x, y) {
  result = x + y  // Implicit global
  return result
}

add(3, 4)
console.log(result)   // prints 7!

function sub(x, y) {
  let val = x + y
  return val
}

sub(42, 2)
//console.log(val)  // ReferenceError
```

Note

Constant variables are not necessarily immutable (their value can change), but they can't be rebound to another object or primitive.

Tip

A good rule of thumb is to use `const` as the default declaration. Use `let` only if needed and try to stay away from `var`.

Scoping

The following function illustrates scoping with the `let` declaration:

```
function scopetest() {
  //console.log(1, x) //ReferenceError
  let x
  console.log(2, x)
  x = 42
  console.log(3, x)
  if (true) {
    //console.log(4, x) // ReferenceError
    let x
    console.log(5, x)
    x = 17
    console.log(6, x)
  }
  console.log(7, x)
}
```

The output is:

```
2 undefined
3 42
5 undefined
6 17
7 42
```

If we use a `var` declaration we see different behavior:

```
function varscopetest() {
  console.log(1, x)
  var x
  console.log(2, x)
  x = 42
  console.log(3, x)
  if (true) {
    console.log(4, x)
    var x
    console.log(5, x)
    x = 17
    console.log(6, x)
  }
  console.log(7, x)
}
```

The output is:

```
1 undefined
2 undefined
3 42
4 42
5 42
6 17
7 17
```

Destructuring

We can pull out variables from a list by *destructuring*:

```
let paul = ['Paul', 1942, 'Bass']
let [name, ,instrument] = paul
```

Default values can be provided during destructuring:

```
let [name, ,instrument='guitar'] = ['Paul', 1942]
```

Copy the list using the *spread* operator:

```
let p2 = [...paul]
```

There is also the notion of object destructuring:

```
let paul = {name: 'Paul', year: 1942};

let {name, inst, year} = paul;

// inst is undefined
```

Default values can also be provided for object destructuring:

```
let paul = {name: 'Paul', year: 1942};

let {name = 'Joe', inst = 'Guitar', year} = paul;
```

We can also rename the properties during object destrucruting:

```
let paul = {name: 'Paul', year: 1942};

let {name: firstname, inst: instrument, year} = paul;
```

...and we can combine renaming with default values:

```
let paul = {name: 'Paul', year: 1942};

let {name: firstname = 'Joe', inst: instrument = 'Guitar', year} = paul;
```

Types

There are two types of data in ES6, primitives and objects. ES6 has six primitive, immutable data types: string, number, boolean, null, undefined, and symbol (new in ECMAScript 2015). When we use the literals we get a primitive. ES6 does *autoboxing* when we invoke a method on them. There are also object wrappers for string, number, boolean, and symbol. They are `String`, `Number`, `Boolean`, and `Symbol` respectively. If you call the constructor with `new`, you will get back on object, to get the primitive value call `.valueOf()`.

`null`

A value that often represents a place where an object will be expected.

```
let count = null
typeof count === "object"
```

NOTE

Even though `null` is a primitive, the result of `typeof` is an object. This is according to the spec [1] (though it is considered a wart).

`undefined`

A property of the global object whose value is the primitive value `undefined`. A variable that has been declared but not assigned has the value of `undefined`:

```
let x  // x is undefined
```

The `typeof` of an `undefined` value is the string `"undefined"`:

```
typeof(x) === 'undefined'  // true
```

Be careful of loose equality and strict equality comparisons with `null`:

```
undefined == null   // true - loose
undefined === null  // false - strict
```

Boolean

A boolean variable can have a value of `true` or `false`.

We can coerce other values to booleans with the `Boolean` wrapper. Most values are truthy:

Truthy and Falsey values

Truthy	Falsey
`true`	`false`
Most objects	`null`
`1`	`0` or `-0`
`"string"`	`""` (empty string)
`[]` (empty list)	`undefined`
`{}` (empty object)	
`Boolean(new Boolean(false))`	

Note calling `new Boolean(obj)` (ie as a constructor) returns a `Boolean` object, whereas calling `Boolean(obj)` (as if it were a function) returns a primitive `true` or `false`. Also, note that coercing any object to a boolean coerces to `true`, even if the internal value was `false`.

A common technique to get primitive boolean values is to use a *double negation* rather than using `Boolean`. This is not necessary in an `if` statement. But, if you want to create a variable that holds a boolean value (or return one), this trick can come in handy. The `!` (not operator) coerces the value to a negated boolean, so if we apply it again, we should get the correct boolean value:

```
"" == false    // true
"" === false   // false
!!"" == false  // true
!!"" === false // true
```

Boolean Properties

Property	Description
`Boolean.length`	`1`
`Boolean.prototype`	The `Boolean` prototype

Boolean Prototype Methods

Method	Description
`b.p.constructor()`	The `Boolean` object
`b.p.toString()`	String with value `"true"` or `"false"`
`b.p.valueOf()`	Primitive boolean value

Objects

ES6 adds the ability to have object keys created from variable names:

```
const name = 'Paul'
const person = { name }  // like name: 'Paul'
```

NOTE

Array spread is an ES6 feature. Object spread is not, though many JS engines support it.

If we want to include properties in another object, we can do a shallow *spread*:

```
const p2 = { ...person, copy: true }
```

In addition there is support for *computed property keys*:

```
const inst = 'Guitar'
const person = {
  // like playsGuitar: true
  ['plays' + inst]: true
}
```

There is also a shorthand for *method definition*:

```
const account = {
   amount: 100,
   // old style
   add: function(amt) {
      this.amount += amt
   },
   // shorthand, no function
   remove(amt) {
    this.amount -= amt
   }
}
```

Typically we would wrap these in a function to create the object:

```
function getAccount(amount) {
  return {
     amount,
     add(amt) {
        this.amount += amt
     },
     // shorthand, no function
     remove(amt) {
      this.amount -= amt
     }
  }
}
```

We can also define properties in an object:

```
function getAccount(amount) {
  return {
     _amount: amount,
     get amount() {
       return this._amount
     },
     set amount(val) {
       this._amount = val
     }
  }
}
```

`Object` can be called as a constructor (with `new`) and as a function. Both behave the same and wrap what they were called with with an object.

Using the methods `Object.defineProperties` and `Object.defineProperty`, we can set properties using *data descriptors* or *accessor descriptors*.

An accessor descriptor allows us to create functions (`get` and `set`) to define member access. It looks like this:

```
{
  configurable: false, // default
  enumerable: false,   // default
  get: getFunc, // function to get prop, default undefined
  set: setFunc, // function to set prop, default undefined
}
```

A data descriptor allows us to create a value for a property and to set whether it is writeable. It looks like this:

```
{
  configurable: false, // default
  enumerable: false,   // default
  value: val           // default undefined
  writeable: false,    // default
}
```

If `configurable` has the value of `false`, then no value besides `writeable` can be changed with `Object.defineProperty`. Also, the property cannot be deleted.

The `enumerable` property determines if the property shows up in `Object.keys()` or a `for ... in` loop.

An example of a accessor descriptor:

```
function Person(fname) {
  let name = fname;
  Object.defineProperty(this, 'name', {
    get: function() {
      // if you say this.name here
      // you will blow the stack
      if (name === 'Richard') {
        return 'Ringo';
      }
      return name;
  },
    set: function(n) {
      name = n
    }
  });
}

let p = new Person('Richard');

console.log(p.name);  // writes 'Ringo'
p.name = 'Fred';
console.log(p.name);  // writes 'Fred'
```

These can be specified on classes as well:

```
class Person {
  constructor(name) {
    this.name = name;
    Object.defineProperty(this, 'name', {
      get: function() {
        // if you say this.name here
```

```
        // you will blow the stack
        if (name === 'Richard') {
          return 'Ringo';
        }
        return name;
      },
      set: function(n) {
        name = n
      }
    });
  }
}
```

The following tables list the properties of an object.

Object Properties

Property	Description
`Object.prototype`	The objects prototype
`Object.prototype.__proto__`	value: `null`

Object Methods

Method	Description
`Object.assign(target, ...sources)`	Copy properties from `sources` into `target`
`Object.create(obj, [prop])`	Create a new object with `prototype` of `obj` and properties from `prop`
`Object.defineProperties( obj, prop)`	Update properties of `obj` from `prop`
`Object.defineProperty( obj, name, desc)`	Create a property named `name` on `obj` with descriptor `desc`
`Object.freeze(obj)`	Prevents future changes to properties (adding of removing) of `obj`. Strict mode throws errors, otherwise silent failure when trying to tweak properties later
`Object.getOwnProperty Descriptor(obj, name)`	Get the descriptor for `name` on `obj`. Can't be in prototype chain.
`Object.getOwnProperty Descriptors( obj)`	Enumerate descriptors on `obj` that aren't in prototype chain.
`Object.getOwnProperty Names(obj)`	Return array of string of string properties found on `obj` that aren't in prototype chain
`Object.getOwnProperty Symbols(obj)`	Return array of symbols of symbol properties found on `obj` that aren't in prototype chain
`Object.getPrototypeOf( obj)`	Return the prototype of `obj`
`Object.is(a, b)`	Boolean whether the values are the same. Doesn't coerce like `==`. Also, unlike `===`, doesn't treat `-0` equal to `+0`, or `NaN` not equal to `NaN`.
`Object.isExtensible(obj)`	Boolean whether the object can have properties added
`Object.isFrozen(obj)`	Boolean whether the object is frozen (frozen is also sealed and non-extensible)
`Object.isSealed(obj)`	Boolean whether the object is sealed (non-extensible, non-removable, but potentially writeable)
`Object.keys(obj)`	Enumerable properties given in `for ... in` loop that are not in prototype chain
`Object.preventExtensions( obj)`	No new properties directly (can be added to prototype), but can remove them
`Object.seal(obj)`	Prevent change of properties on an object. Note that the values can change.
`Object.setPrototypeOf(obj, proto)`	Set the `prototype` property of an object

Object Prototype Methods

Method	Description
`o.p.constructor()`	The `Object` constructor
`o.p.hasOwnProperty(prop)`	Boolean whether `prop` is a direct property of `o` (not in prototype chain)
`o.p.isPrototypeOf(obj)`	Boolean whether `o` exists in `obj`'s prototype chain
`o.p.propertyIs Enumerable(property)`	Boolean whether property is enumerable
`o.p.toLocaleString()`	A string representing the object in locale
`o.p.toString()`	A string representing the object
`o.p.valueOf()`	Return primitive value

1 - http://www.ecma-international.org/ecma-262/6.0/#sec-typeof-operator-runtime-semantics-evaluation

Numbers

NaN

`NaN` is a global property to represent *not a number*. This is the result of certain math failures, such as the square root of a negative number. The function `isNaN` will test whether a value is `NaN`.

`Infinity`

`Infinity` is a global property to represent a very large number. There is also `-Infinity` for for a very large negative values.

You can specify whole number literals as integers, hex, octal, or binary numbers. There is also support for creating float values. See the number types table.

Number types

Type	Example
Integer	`14`
Integer (Hex)	`0xe`
Integer (Octal)	`0o16`
Integer (Binary)	`0b1110`
Float	`14.0`
Float	`1.4e1`

Called as a constructor (`new Number(obj)`), will return a `Number` object.

When called as a function (without `new`), it will perform a type conversion to the primitive.

Number Properties

Property	Description
`Number.EPSILON`	Smallest value between numbers `2.220446049250313e-16`
`Number.MAX_SAFE_INTEGER`	Largest integer `9007199254740991` (`2^53 - 1`)
`Number.MAX_VALUE`	Largest number `1.7976931348623157e+308`
`Number.MIN_SAFE_INTEGER`	Most negative integer `-9007199254740991` (`-(2^53 - 1)`)
`Number.MIN_VALUE`	Smallest number `5e-324`
`Number.NEGATIVE_INFINITY`	Negative overflow `-Infinity`
`Number.NaN`	Not a number value `NaN`
`Number.POSITIVE_INFINITY`	Positive overflow
`Number.name`	value: `Number`
`Number.prototype`	Prototype for `Number` constructor

Number Methods

Method	Description
`n.isFinite(val)`	Test if `val` is finite
`n.isInteger(val)`	Test if `val` is integer
`n.isNaN(val)`	Test if `val` is `NaN`
`n.isSafeInteger(val)`	Test if `val` is integer between safe values
`n.parseFloat(s)`	Convert string, `s` to number (or `NaN`)
`n.parseInt(s, [radix])`	Convert string, `s` to integer (or `NaN`) for given base (`radix`)

Number Prototype Methods

Method	Description
`n.p.constructor()`	
`n.p.toExponential( [numDigits])`	Return a string in exponential notation with `numDigits` of precision
`n.p.toFixed([digits])`	Return a string in fixed-point notation with `digits` of precision
`n.p.toLocaleString([locales, [options]])`	Return a string representation in locale sensitive notation
`n.p.toPrecision([numDigits])`	Return a string in fixed-point or exponential notation with `numDigits` of precision
`n.p.toString([radix])`	Return a string representation. `radix` can be between `2` and `36` for indicating the base
`n.p.valueOf()`	Return the primitive value of the number

`Math` Library

ES6 has a built-in math library to perform common operations.

Math Properties

Property	Description
`Math.E`	value: `2.718281828459045`
`Math.LN10`	value: `2.302585092994046`
`Math.LN2`	value: `0.6931471805599453`
`Math.LOG10E`	value: `0.4342944819032518`
`Math.LOG2E`	value: `1.4426950408889634`
`Math.PI`	value: `3.141592653589793`
`Math.SQRT1_2`	value: `0.7071067811865476`
`Math.SQRT2`	value: `1.4142135623730951`

Math Methods

Method	Description
`Math.abs(n)`	Compute absolute value
`Math.acos(n)`	Compute arccosine
`Math.acosh(n)`	Compute hyperbolic arccosine
`Math.asin(n)`	Compute arcsine
`Math.asinh(n)`	Compute hyperbolic arcsine
`Math.atan(n)`	Compute arctangent
`Math.atan2(y, x)`	Compute arctangent of quotient
`Math.atanh(n)`	Compute hyperbolic arctangent
`Math.cbrt(n)`	Compute cube root
`Math.ceil(n)`	Compute the smallest integer larger than `n`
`Math.clz32(n)`	Compute count of leading zeros
`Math.cos(n)`	Compute cosine
`Math.cosh(n)`	Compute hyperbolic cosine
`Math.exp(x)`	Compute e to the x
`Math.expm1(x)`	Compute e to the x minux 1
`Math.floor(n)`	Compute largest integer smaller than `n`
`Math.fround(n)`	Compute nearest float
`Math.hypot(x, [y], [...)`	Compute hypotenuse (square root of sums)
`Math.imul(x, y)`	Compute integer product
`Math.log(n)`	Compute natural log
`Math.log10(n)`	Compute log base 10
`Math.log1p(n)`	Compute natural log of 1 + `n`
`Math.log2(n)`	Compute log base 2
`Math.max(...)`	Compute maximum
`Math.min(...)`	Compute minimum
`Math.pow(x, y)`	Compute x to the y power
`Math.random()`	Random number between 0 and 1
`Math.round(n)`	Compute nearest integer
`Math.sign(n)`	Return `-1`, `0`, or `1` for negative, zero, or positive value of `n`
`Math.sin(n)`	Compute sine
`Math.sinh(n)`	Compute hyperbolic sine
`Math.sqrt(n)`	Compute square root
`Math.tan(n)`	Compute tangent
`Math.tanh(n)`	Compute hyperbolic tangent
`Math.trunc(b)`	Compute integer value without decimal

Built-in Types

Strings

ES6 strings are series of UTF-16 code units. A string literal can be created with single or double quotes:

```
let n1 = 'Paul'

  let n2 = "John"
```

To make long strings you can use to backslash to signify that the string continues on the following line:

```
let longLine = "Lorum ipsum \

  fooish bar \

  the end"
```

Alternatively the + operator allows for string concatenation:

```
let longLine = "Lorum ipsum " +

  "fooish bar " +

  "the end"
```

Template Literals

Using backticks, you can create *template literals*. These allow for interpolation:

```
let name = 'Paul';

  let instrument = 'bass';



  var `Name: ${name} plays: ${instrument}`
```

Note that template literals can be multi-line:

```
`Starts here

  and ends here`
```

Raw Strings

If you need strings with backslashes in them, you can escape the backslash with a backslash (`/`) or you can use *raw* strings:

```
String.raw `This is a backslash: \

  and this is the newline character: \n`
```

Methods

String Properties

Property	Description
`String.length`	value: `1`
`String.name`	value: `String`
`String.prototype`	Prototype for `String` constructor

Static String Methods

Method	Description
`String.fromCharCode(n1, ...)`	Return string containing characters from Unicode values `n1`
`String.fromCodePoint(n1, ...)`	Return string containing characters from Unicode points `n1`
`String.raw`	Create a raw template literal (follow this by your string surrounded by back ticks)

String Prototype Methods

Method	Description
`s.p.anchor(aName)`	Return `<a name="aName">s</a>`
`s.p.big()`	Return `<big>s</big>`
`s.p.blink()`	Return `<blink>s</blink>`
`s.p.bold()`	Return `<b>s</b>`
`s.p.charAt(idx)`	Return string with character at `idx`. Empty string if invalid index
`s.p.charCodeAt(idx)`	Return integer between 0 and 65535 for the UTF-16 code. `NaN` if invalid index
`s.p.codePointAt(idx)`	Return integer value for Unicode code point. `undefined` if invalid index
`s.p.concat(s1, ...)`	Return concatenation of strings
`s.p.constructor()`	String constructor
`s.p.endsWith(sub, [length])`	Boolean if `s` (limited to `length` size) ends with `sub`
`s.p.fixed()`	Return `<tt>s</tt>`
`s.p.fontcolor(c)`	Return `<font color="c">s</font>`
`s.p.fontsize(num)`	Return `<font size="num">s</font>`
`s.p.includes(sub, [start])`	Boolean if `sub` found in `s` from `start`
`s.p.indexOf(sub, [start])`	Index of `sub` in `s` starting from `start`. `-1` if not found
`s.p.italics()`	Return `<i>s</i>`
`s.p.lastIndexOf(sub, start)`	Return index of `sub` starting from rightmost `start` characters (default `+Infinity`). `-1` if not found
`s.p.link(url)`	Return `<a href="url">s</a>`
`s.p.localeCompare( other, [locale, [option])`	Return `-1`, `0`, or `1` `s` comes before `other`, is equal to, or after
`s.p.match(reg)`	Return an array with the whole match in index 0, remaining entries correspond to parentheses
`s.p.normalize([unf])`	Return the Unicode Normalization Form of a string. `unf` can be `"NFC"`, `"NFD"`, `"NFKC"`, or `"NFKD"`

`s.p.repeat(num)`	Return `s` concatenated with itself `num` times
`s.p.replace(this, that)`	Return a new string with replacing `this` with `that`. `this` can be a regular expression or a string. `that` can be a string or a function that takes `match`, a parameter for every parenthesized match, an index offset for the match, and the original string. It returns a new value.
`s.p.search(reg)`	Return the index where the regular expression first matches `s` or `-1`
`s.p.slice(start, [end])`	Return string sliced at half open interval including `start` and up to be not including `end`. Negative values mean `s.length - val`
`s.p.small()`	Return `<small>s</small>`
`s.p.split([sep, [limit]])`	Return array with string broken into substrings around `sep`. `sep` may be a regular expression. `limit` determines the number of splits in the result. If the regular expression contains parentheses, then the matched portions are also included in the result. Use `s.join` to undo
`s.p.startsWith(sub, [pos])`	Boolean if `s` (beginning at `pos` index) starts with `sub`
`s.p.strike()`	Return `<string>s</strike>`
`s.p.sub()`	Return `<sub>s</sub>`
`s.p.substr(pos, [length])`	Return substring starting at `pos` index (can be negative). `length` is the number of characters to include.
`s.p.substring(start, [end])`	Return slice of string including `start` going up to be not including `end` (half-open). Negative values not allowed.
`s.p.sup()`	Return `<sup>s</sup>`
`s.p.toLocaleLowerCase()`	Return lower case value according to local
`s.p.toLocaleUpperCase()`	Return upper case value according to local
`s.p.toLowerCase()`	Return lower case value
`s.p.toString()`	Return string representation
`s.p.toUpperCase()`	Return upper case value
`s.p.trim()`	Return string with leading and trailing whitespace removed
`s.p.trimLeft()`	Return string with leading whitespace removed
`s.p.trimRight()`	Return string with trailing whitespace removed
`s.p.valueOf()`	Return primitive value of string representation
`s.p[@@iterator]()`	Return an iterator for the string. Calling `.next()` on the iterator returns code points

NOTE

Many of the HTML generating methods create markup that is not compatible with HTML5.

Arrays

ES6 arrays can be created with the literal syntax or by calling the `Array` constructor:

```
let people = ['Paul', 'John', 'George']

  people.push('Ringo')
```

Arrays need not be dense:

```
people[6] = 'Billy'


  console.log(people)

  //["Paul", "John", "George", "Ringo", 6: "Billy"]
```

The `includes` method is useful for checking membership on arrays:

```
people.includes('Yoko')   // true
```

If we need the index number during iteration, the `entries` method gives us a list of index, item pairs:

```
for (let [i, name] of people.entries()) {

    console.log(`${i} - ${name}`);

  }


  // Output

  // 0 - Paul

  // 1 - John

  // 2 - George

  // 3 - Ringo

  // 4 - undefined

  // 5 - undefined

  // 6 - Billy
```

We can do index operations on arrays:

```
let paul = people[0];
```

Note that index operations do not support negative index values:

```
people[-1]; // undefined, not Billy
```

In ES6 we can subclass `Array`:

```
class PostiveArray extends Array {

    push(val) {
```

```
      if (val >  0) {
        super(val);
      }
    }
  }
```

You can *slice* arrays using the `slice` method. Note that the slice returns an array, even if it has a single item. Note that the `slice` method can take negative indices:

```
people.slice(1, 2)  // ['John']
  people.slice(-1)    // ['Billy']
  people.slice(3)     // ["Ringo", undefined × 2, "Billy"]
```

Array Properties

Property	Description
`Array.length`	value: `1`
`Array.name`	value: `Array`
`Array.prototype`	Prototype for `Array` constructor

Array Methods

Method	Description
`Array.from(iter, [func, [this]])`	Return a new `Array` from an iterable. `func` will be called on every item. `this` is a value to use for `this` when executing `func`.
`Array.isArray(item)`	Return boolean if `item` is an `Array`
`Array.of(val, [..., valN])`	Return `Array` with values. Integer values are inserted, whereas `Array(3)`, creates an `Array` with three slots.

Array Prototype Methods

Method	Description
`a.p.concat(val, [..., valN])`	Return a new `Array`, with values inserted. If value is array, the items are appended
`a.p.copyWithin(target, [start, [end]])`	Return `a` mutated with items from `start` to `end` shallow copied into index `target`
`a.p.entries()`	Return array iterator
`a.p.every(func, [this])`	Return boolean if `func(item, [idx, [a]])` is true for every item in array (`idx` is the index, and `a` is the array). `this` is the value of `this` for the function
`a.p.fill(val, [start, [end]])`	Return `a` mutated, with `val` inserted from `start` index up to but not including `end` index. Index values may be negative
`a.p.filter(func, [this])`	Return a new array of items from `a` where predicate `func(item, [idx, [a]])` is true. `this` is the value of `this` for the function
`a.p.find(func, [this])`	Return first item (or `undefined`) from `a` where predicate `func(item, [idx, [a]])` is true. `this` is the value of `this` for the function
`a.p.findIndex(func, [this])`	Return first index (or `-1`) from `a` where predicate `func(item, [idx, [a]])` is true. `this` is the value of `this` for the function
`a.p.forEach(func, [this])`	Return `undefined`. Apply `func(item, [idx, [a]])` to every item of `a`. `this` is the value of `this` for the function
`a.p.includes(val, [start])`	Return a boolean if `a` contains `val` starting from `start` index
`a.p.indexOf(val, [start])`	Return first index (or `-1`) of `val` in `a`, starting from `start` index
`a.p.join([sep])`	Return string with `sep` (default `','`) inserted between items
`a.p.keys()`	Return iterator of index values (doesn't skip sparse values)
`a.p.lastIndexOf(val, [start])`	Return last index (or `-1`) of `val` in `a`, searching backwards from `start` index
`a.p.map(func, [this])`	Return new array with `func(item, [idx, [a]])` called on every item of `a`. `this` is the value of `this` for the function
`a.p.pop()`	Return last item (or `undefined`) of `a` (mutates `a`)
`a.p.push(val, [..., valN])`	Return new length of `a`. Add values to end of `a`
`a.p.reduce(func, [init])`	Return results of reduction. Call `func(accumulator, val, idx, a)` for every item. If `init` is provided, `accumulator` is set to it initially, otherwise `accumulator` is first item of `a`.
`a.p.reduceRight(func, [init])`	Return results of reduction applied backwards
`a.p.reverse()`	Return and mutate `a` in reverse order
`a.p.shift()`	Return and remove first item from `a` (mutates `a`)
`a.p.slice([start, [end]])`	Return shallow copy of `a` from `start` up to but not including `end`. Negative index values allowed
`a.p.some(func, [this])`	Return boolean if `func(item, [idx, [a]])` is true for any item in array (`idx` is the index, and `a` is the array). `this` is the value of `this` for the function
`a.p.sort([func])`	Return and mutate sorted `a`. Can use `func(a, b)` which returns `-1`, `0`, or `1`
`a.p.splice(start, [deleteCount, [item1, ..., itemN]])`	Return array with deleted objects. Mutate `a` at index `start`, remove `deleteCount` items, and insert `items`.
`a.p.toLocaleString( [locales, [options]])`	Return a string representing array in locales
`a.p.toString()`	Return a string representing array in locales
`a.p.unshift([item1, ... itemN])`	Return length of `a`. Mutate `a` by inserting elements in front of `a`. (Will be in same order as appearing in call)
`a.p.values()`	Return iterator with items in `a`

ArrayBuffers

An `ArrayBuffer` holds generic byte based data. To manipulate the contents, we point a view (typed arrays or data views) at certain locations:

```
let ab = new ArrayBuffer(3) let year = new Uint16Array(ab, 0, 1)

  let age = new Uint8Array(ab, 2, 1)

  year[0] = 1942

  age[0] = 42
```

ArrayBuffer Properties

Property	Description
`ArrayBuffer.length`	`1`
`ArrayBuffer.prototype`	The `ArrayBuffer` prototype

ArrayBuffer Prototype Methods

Method	Description
`a.p.constructor()`	The `ArrayBuffer` object
`a.p.slice(begin, [end])`	Return a new `ArrayBuffer` with copy of `a` from `begin` up to but not including `n`

TypedArrays

ES6 supports various flavors of *typed arrays*. These hold binary data, rather than ES6 objects:

```
let size = 3;

  let primes = new Int8Array(size);

  primes[0] = 2;

  primes[1] = 3;

  primes[2] = 5;  // size now 3

  primes[3] = 7;  // full! ignored

  console.log(primes) // Int8Array [ 2, 3, 5 ]
```

There are a few differences from normal arrays:

- Items have the same type
- The array is contiguous
- It is initialized with zeros

To put a typed array into a normal array, we can use the *spread* operator:

```
let normal = [...primes]
```

Typed Arrays

Type	Size (bytes)	Desctription	C type
`Int8Array`	1	signed integer	`int8_t`
`Uint8Array`	1	unsigned integer	`uint8_t`
`Uint8ClampedArray`	1	unsigned integer	`uint8_t`
`Int16Array`	2	signed integer	`int16_t`
`Uint16Array`	2	unsigned integer	`unint16_t`
`Int32Array`	4	signed integer	`int32_t`
`Uint32Array`	4	unsigned integer	`unint32_t`
`Float32Array`	4	32 bit floating point	`float`
`Float64Array`	8	64 bit floating point	`float`

The following `Array` methods are missing: `push`, `pop`, `shift`, `splice`, and `unshift`.

There are also two extra methods found on `TypedArrays`

TypedArray Prototype Methods

Method	Description
`t.p.set(array, [offset])`	Return `undefined`. Copy `array` into `t` at `offset` location
`t.p.subarray(start, [end])`	Return `TypeArray` with view of data in `t` starting at position `start` up to but not including `end`. Note that the `t` and the new object share the data.

Data Views

A `DataView` is another interface for interacting with an `ArrayBuffer`. If you need control over endianness, you should use this rather than a typed array.

```
let buf = new ArrayBuffer(2) let dv = new DataView(buf)

  let littleEndian = true;

  let offset = 0

  dv.setInt16(offset, 512, littleEndian)


  let le = dv.getInt16(offset, littleEndian) //512


  let be = dv.getInt16(offset) // Big endian 2
```

The constructor supports a buffer and option offset and length:

```
new DataView(buffer, [offset, [length]])
```

DataView Properties

Property	Description
`DataView.name`	`DataView`
`DataView.prototype`	`DataView` constructor
`DataView.prototype.buffer`	The underlying `ArrayBuffer`
`DataView.prototype.byteLength`	The length of the view
`DataView.prototype.byteOffset`	The offset of the view

DataView Prototype Methods

Method	Description
`d.p.getFloat32(offset, [littleEndian])`	Retrieve signed 32-bit float from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getFloat64(offset, [littleEndian])`	Retrieve signed 64-bit float from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getInt16(offset, [littleEndian])`	Retrieve signed 16-bit integer from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getInt32(offset, [littleEndian])`	Retrieve signed 32-bit integer from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getInt8(offset, [littleEndian])`	Retrieve signed 8-bit integer from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getUint16(offset, [littleEndian])`	Retrieve unsigned 16-bit integer from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getUint32(offset, [littleEndian])`	Retrieve unsigned 32-bit integer from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.getUint8(offset, [littleEndian])`	Retrieve unsigned 8-bit integer from `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setFloat32(offset, value, [littleEndian])`	Set signed 32-bit float `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setFloat64(offset, value, [littleEndian])`	Set signed 64-bit float `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setInt16(offset, value, [littleEndian])`	Set signed 16-bit integer `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setInt32(offset, value, [littleEndian])`	Set signed 32-bit integer `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setInt8(offset, value, [littleEndian])`	Set signed 8-bit integer `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setUint16(offset, value, [littleEndian])`	Set unsigned 16-bit integer `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setUint32(offset, value, [littleEndian])`	Set unsigned 32-bit integer `value` at `offset`. If `littleEndian` is `true` use litte endian format
`d.p.setUint8(offset, value, [littleEndian])`	Set unsigned 8-bit integer `value` at `offset`. If `littleEndian` is `true` use litte endian format

Date

Options for `Date`. To get the current time simply use:

```
let now = new Date();
```

If an integer is provided, you will get the seconds from the Unix epoch:

```
let msPast1970 = 1

 let groovyTime = new Date(msPast1970)



 // Wed Dec 31 1969 17:00:00 GMT-0700 (MST)
```

The ES6 spec only supports a variant of ISO8601, but in practice a string in RFC 2822/1123 is also supported:

```
let modern = new Date("Tue, 14 Mar 2017 14:59:59 GMT")
```

Finally, the `Date` constructor allows us to specify the year, month, date, hours, minutes, seconds, and milliseconds:

```
let piDay = new Date(2017, 3, 14)
```

Dates created with the constructor are in the local time. To create a `Date` in UTC, use the `Date.UTC` method.

NOTE

An RFC 2822/RFC 1123 string is a human readable string that looks like:

```
"Tue, 14 Mar 2017 14:59:59 GMT"
```

The `toUTCString` method will give you this string.

NOTE

ISO 8601 in ES6 is specified like this:

```
YYYY-MM-DDTHH:mm:ss.sssZ
```

There is a `toISOString` method on `Date` that will return this format.

Date Properties

Property	Description
`Date.name`	`Date`
`Date.prototype`	`Date` constructor

Date Methods

Method	Description
`d.UTC(year, month, [day, [hour, [minute, [second, [millisecond]]]]])`	Return milliseconds since Unix epoch from UTC time specified
`d.now()`	Return milliseconds since Unix epoch
`d.parse(str)`	Return milliseconds since Unix epoch for ISO 8601 string

Date Prototype Methods

Method	Description
`d.p.getDate()`	Return day of month, number between `1` and `31`
`d.p.getDay()`	Return day of the week (`0` is Sunday).
`d.p.getFullYear()`	Return year, number between `0` and `9999`
`d.p.getHours()`	Return hour, number between `0` and `23`
`d.p.getMilliseconds()`	Return milliseconds, number between `0` and `999`
`d.p.getMinutes()`	Return minutes, number between `0` and `59`
`d.p.getMonth()`	Return month, number between `0` (Jan) and `11` (Dec)
`d.p.getSeconds()`	Return seconds, number between `0` and `59`
`d.p.getTime()`	Return milliseconds since Unix epoch
`d.p.getTimezoneOffset()`	Return timezone offset in minutes
`d.p.getUTCDate()`	Return UTC day of month, number between `1` and `31`
`d.p.getUTCDay()`	Return UTC day of the week (`0` is Sunday).
`d.p.getUTCFullYear()`	Return UTC year, number between `0` and `9999`
`d.p.getUTCHours()`	Return hour, number between `0` and `23`
`d.p.getUTCMilliseconds()`	Return UTC milliseconds, number between `0` and `999`
`d.p.getUTCMinutes()`	Return UTCminutes, number between `0` and `59`
`d.p.getUTCMonth()`	Return UTCmonth, number between `0` (Jan) and `11` (Dec)
`d.p.getUTCSeconds()`	Return UTC seconds, number between `0` and `59`
`d.p.getYear()`	Broken year implementation, use `getFullYear`
`d.p.setDate(num)`	Return milliseconds after Unix epoch after mutating day of month
`d.p.setFullYear(year, [month, [day]])`	Return milliseconds after Unix epoch after mutating day values
`d.p.setHours(hours, [min, [sec, [ms]]])`	Return milliseconds after Unix epoch after mutating time values
`d.p.setMilliseconds(ms)`	Return milliseconds after Unix epoch after mutating ms value
`d.p.setMinutes(min, [sec, [ms]])`	Return milliseconds after Unix epoch after mutating time values
`d.p.setMonth(month, [day])`	Return milliseconds after Unix epoch after mutating day values
`d.p.setSeconds(sec, [ms])`	Return milliseconds after Unix epoch after mutating time values
`d.p.setTime(epoch)`	

	Return milliseconds after Unix epoch after mutating time value
`d.p.setUTCDate(num)`	Return milliseconds after Unix epoch after mutating day of month
`d.p.setUTCFullYear(year, [month, [day]])`	Return milliseconds after Unix epoch after mutating day values
`d.p.setUTCHours(hours, [min, [sec, [ms]]])`	Return milliseconds after Unix epoch after mutating time values
`d.p.setUTCMilliseconds( ms)`	Return milliseconds after Unix epoch after mutating ms value
`d.p.setUTCMinutes(min, [sec, [ms]])`	Return milliseconds after Unix epoch after mutating time values
`d.p.setUTCMonth(month, [day])`	Return milliseconds after Unix epoch after mutating day values
`d.p.setUTCSeconds(sec, [ms])`	Return milliseconds after Unix epoch after mutating time values
`d.p.setYear(year)`	Broken year implementation, use `setFullYear`
`d.p.toDateString()`	Return human readable of date in American English
`d.p.toGMTString()`	Broken, use `toUTCString`
`d.p.toISOString()`	Return date string in ISO 8601 form
`d.p.toJSON()`	Return date JSON string form (ISO 8601)
`d.p.toLocaleDateString( [locales, [options]])`	Return string of date portion in locale
`d.p.toLocaleString( [locales, [options]])`	Return string of date and time in locale
`d.p.toLocaleTimeString( [locales, [options]])`	Return string of time in locale
`d.p.toString()`	Return a string representation in American English
`d.p.toTimeString()`	Return a string representation of time portion in American English
`d.p.toUTCString()`	Return (usually RFC-1123 formatted) version of string in UTC timezone
`d.p.valueOf()`	Return milliseconds after Unix epoch

Maps

Maps do not have prototypes (like objects do) that could collide with your keys.

Objects only support strings or symbols as keys, whereas maps support any type for keys (functions, objects, or primitives). Another benefit of maps is that you can easily get the length with the `size` property. To get the length of an object, you need to iterate over it.

Should you use an object or a map? If you need record type data, use an object. For hashlike collections that you need to mutate and iterate over, choose a map.

A `Map` can be created simply by calling the constructor, or by passing an iterable of key value pairs.

```
let instruments = new Map([['Paul', 'Bass'], ['John', 'Guitar']]);

 instruments.set('George', 'Guitar');


 instruments.has("Ringo");  // false for (let [name, inst] of instruments) {

   console.log(`${name} - ${inst}`);

 }
```

Map Properties

Property	Description
`Map.name`	`Map`
`Map.prototype`	Constructor prototype for `Map`
`Map.prototype.size`	Return number of items in map

Map Prototype Methods

Method	Description
`m.p.clear()`	Return `undefined`. Remove all items from `m` (mutates `m`)
`m.p.delete(key)`	Return boolean if `key` was in `m`. Mutates `m` and removes it
`m.p.entries()`	Return iterator for array of key, value pairs
`m.p.forEach(func, [this])`	Return `undefined`. Apply `func(value, key, m)` to every key and value in `m`
`m.p.get(key)`	Return value or `undefined`
`m.p.has(key)`	Return boolean if `key` found in `m`
`m.p.keys()`	Return iterator of keys in `m` in order of insertion
`m.p.set(key, value)`	Return `m`, mutating `m` with `value` for `key`
`m.p.values()`	Return iterator for values in order of insertion

WeakMaps

`WeakMaps` allow you to track objects until they are garbage collected. The keys don't create references, so data may disappear from weakmap if the key happens to be garbage collected (or if its containing object is collected).

The constructor has the same interface as `Map`, you can create an empty `WeakMap` by provided no arguments, or you can pass in an iterable of key value pairs.

WeakMap Properties

Property	Description
`WeakMap.name`	`WeakMap`
`WeakMap.prototype`	Constructor prototype for `WeakMap`

WeakMap Prototype Methods

Method	Description
`w.p.delete(key)`	Return `true` if `key` existed and was removed, else `false`
`w.p.get(key)`	Return value for `key` or `undefined` if missing
`w.p.has(key)`	Return boolean if `key` found in `w`
`w.p.set(key, value)`	Return `w` after mutating it with new `key` and `value`

Sets

A set is a mutable unordered collection that cannot contain duplicates. Sets are used to remove duplicates and test for membership. You can make an empty `Set` by calling the constructor with no arguments. If you wish to create a `Set` from an iterable, pass that into the constructor:

```
let digits = new Set([0, 1, 1, 2, 3, 4, 5, 6, 7, 8 , 9]);


  digits.has(9);  // true


  let odd = new Set([1, 3, 5, 7, 9]);
  let prime = new Set([2, 3, 5, 7]);
```

Set operations are not provided. Here is an example of adding difference:

```
Set.prototype.difference = function(other) {
    let result = new Set(this);
    for (let val of other) {
      result.delete(val);
    }
    return result;
  }


  let even = digits.difference(odd);
  console.log(even);  // Set { 0, 2, 4, 6, 8 }
```

Set Properties

Property	Description
`Set.name`	`Set`
`Set.prototype`	Constructor prototype for `Set`
`Set.prototype.size`	Return size of set

Set Prototype Methods

Method	Description
`s.p.add(item)`	Return `s`. Add `item` to `s` (mutating)
`s.p.clear()`	Return `undefined`. Removes (mutating) all items from `s`
`s.p.delete(item)`	Return value if deleted, otherwise returns `false` (mutating)
`s.p.entries()`	Return iterator of `item`, `item` pairs (both the same) for every item of `s` (same interface as `Map`)
`s.p.forEach(func, [this])`	Return `undefined`. Apply `func(item, item, s)` to every item in `s`. Same interface as `Array` and `Map`
`s.p.has(item)`	Return boolean if `item` found in `s`
`s.p.values()`	Return iterator for items in insertion order

Weak Set

Weak sets are collections of objects, and not any type. We cannot enumerate over them. Objects may spontaneously disappear from them when they are garbage collected.

The `WeakSet` constructor has the same interace as `Set` (no arguments or an iterable).

WeakSet Properties

Property	Description
`WeakSet.name`	`WeakSet`
`WeakSet.prototype`	Prototype for `WeakSet`

WeakSet Prototype Methods

Method	Description
`w.p.add(item)`	Return `w`, inserts `item` into `w`
`w.p.delete(item)`	Return `true` if `item` was in `w` (also removes it), otherwise return `false`
`w.p.has(item)`	Return boolean if `item` in `w`

Proxies

A proxy allows you to create custom behavior for basic operations (getting/setting properties, calling functions, looping over values, decorating, etc). We use a *handler* to configure *traps* for a *target*.

The constructor takes two arguments, a target, and a handler:

```
const handler = {
    set: function(obj, prop, value) {
      if (prop === 'month') {
        if (value < 1 || value > 12) {
          throw RangeError("Month must be between 1 & 12") }
      }
      obj[prop] = value
      // need to return true if successful
      return true
    }
  }

  const cal = new Proxy({}, handler)

  cal.month = 12
  console.log(cal.month)  // 12
  cal.month = 0  // RangeError
```

Proxy Methods

PROPERTY	DESCRIPTION
`Proxy.revocable(target, handler)`	Create a revocable proxy. When `revoke` is called, proxy throws `TypeError`

Reflection

The `Reflect` object allows you to inspect objects. `Reflect` is not a constructor, all the methods are static.

Math Methods

Method	Description
`Reflect.apply(obj, this, args)`	Return result of `obj(...args)` with `this` value
`Reflect.construct(obj, args)`	Return result of `new obj(...args)`
`Reflect.defineProperty(obj, key, descriptor)`	Like `Object.defineProperty( obj, key, descriptor)`, but returns `Boolean`
`Reflect.deleteProperty(obj, key)`	Remove `key` from `obj`, returns `Boolean`
`Reflect.get(obj, key)`	Return `obj[key]`
`Reflect.getOwnProperty Descriptor(obj, key)`	Return property descriptor of `obj[key]`
`Reflect.getPrototypeOf(obj)`	Return prototype for `obj` or `null` (if no inherited properties)
`Reflect.has(obj, key)`	Return `key in obj`
`Reflect.isExtensible(obj)`	Return `Boolean` if you can add new properties to `obj`
`Reflect.ownKeys(obj)`	Return `Array` of keys in `obj`
`Reflect.preventExtensions( obj)`	Disallow extensions on `obj`, return `Boolean` if successful
`Reflect.set(obj, key, value, [this])`	Return `Boolean` if successful setting property `key` on `obj`.

Symbols

ES6 introduced a new primitive type, Symbol. They have string-like properties (immutable, can't set properties on them, can be property names). They also have object-like behavior (unique from others even if the description is the same).

Symbols are unique values that can be used for property keys without collisions.

They must be accessed using an index operation (square brackets) and not dot notation. The are also not iterated over in a `for ... in` loop. To retrieve them we need to use `Object.getOwnPropertySymbols` or `Reflect.ownKeys`.

The constructor takes an optional description argument:

```
Symbol('name') == Symbol('name')   // false Symbol('name') === Symbol('name')  // false
```

NOTE

`Symbol` is not a constructor, and a `TypeError` will be raised if you try to use it as one.

Symbol Properties

PROPERTY	DESCRIPTION
`Symbol.hasInstance`	Used to define class behavior for `instanceof`. Define method as `static [Symbol.hasInstance](instance) ...`
`Symbol.isConcatSpreadable`	Used to define class behavior for `Array.concat`. Set to `true` if items are spread (or flattened).
`Symbol.iterator`	Used to define class behavior for `for...of`. Should follow iteration protocol. Can be a generator
`Symbol.match`	Used to define class behavior for responding as a regular expression in `String` methods: `startsWith`, `endsWith`, `includes`
`Symbol.name`	value: `Symbol`
`Symbol.prototype`	The prototype for `Symbol`
`Symbol.replace`	Used to define class behavior for responding to `String.p.replace` method
`Symbol.search`	Used to define class behavior for responding to `String.p.search` method
`Symbol.species`	Used to define class behavior for which constructor to use when creating derived objects
`Symbol.split`	Used to define class behavior for responding to `String.p.split` method
`Symbol.toPrimitive`	Used to define class behavior for responding to coercion. Define method as `static [Symbol.toPrimitive] (hint) ...`, where `hint` can be `'number'`, `'string'`, or `'default'`
`Symbol.toStringTag`	Used to define class behavior for responding to `Object.p.toString` method
`Symbol.unscopables`	Used to define class behavior in `with` statement. Should be set to an object mapping properties to boolean value if they are not visible in `with`. (ie `true` means throw `ReferenceError`)

Symbol Methods

METHOD	DESCRIPTION
`Symbol.for(key)`	Return symbol in global registry for `key`, other create symbol and return it.
`Symbol.keyFor(symbol)`	Return string value for symbol in global registry, `undefined` if symbol not in registry

Symbol Prototype Methods

METHOD	DESCRIPTION
`s.p.toString()`	Return string representation for symbol
`s.p.valueOf()`	Return primitive value (symbol) of a symbol

Built-in Functions

Built-in Functions

METHOD	DESCRIPTION
`eval(str)`	Evaluate code found in str
`isFinite(val)`	Return `false` if `val` is `Infinity`, `-Infinity`, or `NaN`, else `true`
`isNaN(val)`	Return `true` if `val` is `NaN`, else `False`
`parseFloat(str)`	Return float if `str` can be converted to a number, else `NaN`
`parseInt(val, radix)`	Return integer if `str` can be converted to an integer, else `NaN`. It ignores characters that are not numbers in `radix`
`decodeURI(uri)`	Return the unencoded version of the string. Should be used on full URI
`decodeURIComponent(str)`	Return the unencoded version of the string. Should be used on parts of URI
`encodeURI(uri)`	Return the encoded version of the URI. Should be used on full URI
`encodeURIComponent(uri)`	Return the encoded version of the URI. Should be used on parts of URI

Unicode

If we have a Unicode glyph, we can include that directly:

```
let xSq = 'x²';
```

Alternatively, we can use the Unicode code point to specify a Unicode character:

```
let xSq2 = 'x\u{b2}';
```

If we have exactly four hexadecimal digits we can escape like this:

```
let xSq3 = 'x\u00b2';
```

We can get a code point using the `codePointAt` string method:

```
'x\u{b2}'.codePointAt(1)   // 178
```

To convert a code point back to a string using the `fromCodePoint` static method:

```
String.fromCodePoint(178)  // "²"
```

If we use the `/u` flag in a regular expression, we can search for Unicode characters, which will handle surrogate pairs.

Functions

Functions are easy to define, we simply give them a name, the parameters they accept and a body:

```
function add(x, y) {
  return x + y
}

let six = add(10, -4)
```

The arguments are stored in an implicit variable, `arguments`. We can invoke functions with any number of arguments:

```
function add2() {
  let res = 0
  for (let i of arguments) {
    res += i
  }
  return res
}

let five = add2(2, 3)
```

Default Arguments

If we want to have a default value for an argument use a = to specify it immediately following the argument:

```
function addN(x, n=42) {
  return x + n;
}

let forty = addN(-2)
let seven = addN(3, 4)
```

Variable Parameters

Using . . . (*rest*) turns the remaining parameters into an array:

```
function add_many(...args) {
  // args is an Array
  let result = 0;
  for (const val of args) {
    result += val;
  }
  return result;
}
```

Again, because ES6 provides the `arguments` object, we can also create a function that accepts variable parameters like this:

```
function add_many() {
  let result = 0;
  for (const val of arguments) {
    result += val;
  }
  return result;
}
```

Calling Functions

You can use `...` to *spread* an array into arguments:

```
add_many(...[1, 42., 7])
```

The `bind` Method

Functions have a method, `bind`, that allows you to set `this` and any other parameters. This essentially allows you to *partial* the function:

```
function mul(a, b) { return a * b }

let double = mul.bind( undefined, 2 )
let triple = mul.bind( undefined, 3 )

console.log(triple(2))   // 6
```

The first parameter to bind is the value passed for this. The rest are the arguments for the function. If you want a callback to use the parent's `this`, you can call bind on the function passing in the parent's `this`. Alternatively, you can use an arrow function.

Arrow functions

ES6 introduced anonymous *arrow* functions. There are a few differences with arrow functions:

- Implicit return
- `this` is not rebound
- Cannot be generators

The second feature makes them nice for callbacks and handlers, but not a good choice for methods. We can write:

```
function add2(val) {
  return val + 2
}
```

As:

```
let add2 = (val) => (val + 2)
```

The => is called a *fat arrow*. Since, this is a one line function, we can remove the curly braces and take advantage of the implicit return.

Note that the parentheses can be removed if you only have a single parameter and are inlining the function:

```
let vals = [1, 2, 3]
console.log(vals.map(v=>v*2))
```

If we want a multiline arrow function, then remove the `function`, and add =>:

```
function add(x, y) {
  let res = x + y
  return res
}
```

becomes:

```
let add = (x,y) => {
  let res = x + y
  return res
}
```

Tail Call

If you perform a recursive call in the last position of a function, that is called *tail call*. ES6 will allow you to do this without growing the stack:

```
function fib(n){
  if (n == 0) {
    return 0;
  }
  return n + fib(n-1);
}
```

Note

Some implementations might not support this. This fails with Node 7.7 and Chrome 56 with `fib(100000)`.

Classes

ES6 INTRODUCED CLASS WHICH IS SYNTACTIC SUGAR AROUND CREATING OBJECTS WITH FUNCTIONS. ES6 CLASSES support prototype inheritance, calls to parent classes via `super`, instance methods, and static methods:

```
class Bike {
  constructor(wheelSize, gearRatio) {
    this._size = wheelSize;
    this.ratio = gearRatio;
  }

  get size() { return this._size }
  set size(val) { this._size = val }

  gearInches() {
    return this.ratio * this.size;
  }
}
```

Note that when you create a new instance of a `class`, you need to use `new`:

```
let bike = new Bike(26, 34/13)
bike.gearInches()
```

NOTE

Classes in ES6 are not *hoisted*. This means that you can't use a class until after you defined it. Functions, are hoisted, and you can use them anywhere in the scope of the code that defines them.

Prior to ES6, we could only create objects from functions:

```
function Bike2(wheelSize, gearRatio) {
  this.size = wheelSize;
  this.ratio = gearRatio;
}

Bike2.prototype.gearInches = function() {
  return this.ratio * this.size
}

let b = new Bike2(27, 33/11);
console.log(b.gearInches());
```

NOTE

The method is added after the fact to the prototype, so the instances can share the method. We could define the method in the function, but then every instance would get its own copy.

ES6 just provides an arguably cleaner syntax for this.

Subclasses

One thing to be aware of with subclasses is that they should call `super`. Because ES6 is just syntactic sugar, if we don't call `super`, we won't have the prototypes, and we can't create an instance without the prototypes. As a result, `this` is undefined until `super` is called. If you don't call super you should return `Object.create(...)`.

```
class Tandem extends Bike {
  constructor(wheelSize, rings, cogs) {
    let ratio = rings[0] / cogs;
    super(wheelSize, ratio);
    this.cogs = cogs;
    this.rings = rings;

  }

  shift(ringIdx, cogIdx) {
    this.ratio = this.rings[ringIdx] /
      this.cogs[cogIdx];
  }
}

let tan = new Tandem(26, [42, 36], [24, 20, 15, 11])
tan.shift(1, 2)
console.log(tan.gearInches())
```

Static Methods

A *static method* is a method called directly on the class, not on the instance.

```
class Recumbent extends Bike {
  static isFast() {
    return true;
  }
}

Recumbent.isFast();  // true

rec = new Recumbent(20, 4);
rec.isFast();  // TypeError
```

Object Literal

We can also create instances using object literals, though in practice this leads to a lot of code duplication:

```
let size = 20;
 let gearRatio = 2;

 let bike = {
   __proto__: protoObj,
   ['__proto__']: otherObj,
   size, // same as `size: size`
   ratio: gearRatio,
   gearInches() {
     return this.size * this.ratio;
   }
  // dynamic properties
  [ 'prop_' + (() => "foo")() ]: "foo"
}
```

Operators

Assignment

Built-in Operators

Operator	Description
`=`	Assignment
`+`	Addition, , unary plus (coerce to number), concatenation (string)
`++`	Increment
`-`	Subtraction, unary negation (coerce to number)
`--`	Decrement
`*`	Multiplication
`/`	Division
`%`	Remainder (modulus)
`**`	Power
`<<`	Left shift
`>>`	Right shift
`<<<`	Unsigned left shift
`>>>`	Unsigned right shift
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`~`	Bitwise NOT
`&&`	Logical AND
`||`	Logical OR
`!`	Logical NOT
`,`	Comma operator, evaluates all operands and returns last
`delete X`	Delete an object, property, index
`typeof`	Return string indicating type of operand
`void`	Create an expression without a return value
`in`	Return boolean if property in object
`instanceof`	Return boolean if object instance of type
`new`	Create a new instance of a type
`...`	*Spread* sequence into array or parameters

Conditionals

ES6 has an `if` statement with zero or more `else if` statements, and an optional `else` statement at the end:

```
let grade = 72;

function letter_grade(grade) {
  if (grade > 90) {
    return 'A';
  }
  else if (grade > 80) {
    return 'B';
  }
  else if (grade > 70) {
    return 'C';
  }
  else {
    return 'D';
  }
}

letter_grade(grade);  // 'C'
```

ES6 supports the following tests: `>`, `>=`, `<`, `<=`, `==`, `!=`, `===`, and `!==`. For boolean operators use `&&`, `||`, and `!` for and, or, and not respectively.

For `==` and `!=`, ES6 tries to compare numeric values of the operands if they have differing types, hence:

```
'3' == 3  // true
```

If this bothers you (and it should), use the *strict* equality operators (`===` and `!==`):

```
'3' === 3  // false
```

Short Circuiting

The `and` statement will *short circuit* if it evaluates to false:

```
0 && 1/0  // 0
```

Likewise, the `or` statement will short circuit when something evaluates to true:

```
1 || 1/0  // 1
```

Ternary Operator

ES6 has a ternary operator. Instead of writing:

```
let last
if (band == 'Beatles') {
  last = 'Lennon'
}
else {
  last = 'Jones'
}
```

We can write:

```
let last = (band == 'Beatles) ? 'Lennon' : 'Jones';
```

Switch

ES6 supports the switch statement:

```
function strings(inst) {
  switch(inst) {
    case 'guitar':
      return 6;
    case 'violin':
      return 4;
    default:
      return 1;
  }
}

strings('violin');  // 4
```

Looping

There are various ways to iterate:

- `for ... in` - Iterates over the properties of an object. This only walks through properties that have `[[Enumerable]]` set to `true`.
- `for ... of` - Iterates over the items of a collection. Any object which has the `[Symbol.iterator]` property can be iterated with this method.
- `forEach` is a method on the `Array` object. It takes a callback that is invoked for every item of the array.

There is also a `while` loop:

```
let num = 3
while (num > 0) {
  console.log(num)
  num -= 1
}
console.log('Blastoff!')
```

And a `do ... while` loop:

```
let num = 3
do {
  console.log(num)
  num -= 1
} while (num > 0)
console.log('Blastoff!')
```

Iteration

We can make a class that knows how to iterate. We have to provide a method for `[Symbol.iterator]`, and the result of that method needs to have a `next` method.

Here is an example of creating a class for iteration:

```
class Fib {
  constructor() {
    this.val1 = 1;
    this.val2 = 1;
  }

  [Symbol.iterator]() {
    return this;  // something with next
  }

  next() {
    [this.val1, this.val2] = [this.val2, this.val1 + this.val2];
    return {done: false, value: this.val1};
  }
}
```

The result of `next` should be an object that indicates whether looping is finished in the `done` property, and returns the item of iteration in the `value` property.

And here we use it in a `for .. of` loop:

```
for (var val of new Fib()) {
  console.log(val);
  if (val > 5) {
    break;
  }
}
```

We can also loop with an object literal:

```
let fib = {
  [Symbol.iterator]() {
    let val1 = 1;
    let val2 = 1;
    return {
      next() {
        [val1, val2] = [val2, val1 + val2];
        return { value: val1, done: false}
      }
    }
  }
}
```

Use the iterator in a loop:

```
for (var val of fib) {
  console.log(val);
  if (val > 5) {
    break;
  }
}
```

Exceptions

ES6 ALLOWS US TO HANDLE EXCEPTIONS SHOULD THEY OCCUR:

```
try {
   // code missing
}
catch(e) {
  // handle any exception
}
```

If there is a `finally` statement, it executes after the other blocks, regardless of whether and exception occurred:

```
try {
  // code missing
}
catch(e) {
  // handle any exception
}
finally {
  // run after either block
}
```

Throwing Errors

ES6 allows us to throw errors as well:

```
throw new Error("Some error");
```

Error Types

There are various built-in error types:

- `EvalError` - Not used in ES6, available for backward compatibility
- `RangeError` - A value that is out of the set of allowed values
- `ReferenceError` - Error for referring to a non valid reference (`let val = badRef`)
- `SyntaxError` - Error when syntax is incorrect (`foo bar`)
- `TypeError` - Error when type of value is incorrect (`undefined.junk()`)
- `URIError` - Error when URI encoding/decoding goes awry (`decodeURI('%2')`)

Generators

Iterators that use `function*` and `yield`, rather than normal `function` and `return` are *generators*. They generate values on the fly as they are iterated over. Following a `yield` statement, the state of the function is frozen.

```
let fibGen = {
  [Symbol.iterator]: function*() {
    let val1 = 1;
    let val2 = 2;
    while (true) {
      yield val1;
      [val1, val2] = [val2, val1 + val2];
    }
  }
}
```

Using a generator:

```
for (var val of fibGen) {
    console.log(val);
    if (val > 5) {
        break;
    }
}
```

Modules

A MODULE IS A JAVASCRIPT FILE. TO USE OBJECT IN OTHER FILES, WE NEED TO *EXPORT* THE OBJECTS. HERE WE create a `fib.js` file and export the generator:

```
// js/fib.js

export let fibGen = {
  [Symbol.iterator]: function*() {
    let val1 = 1;
    let val2 = 2;
    while (true) {
      yield val1;
      [val1, val2] = [val2, val1 + val2];
    }
  }
}

export let takeN = {
  [Symbol.iterator]: function*(seq, n) {
    let count = 0;
    for (let val of seq ) {
      if (count >= n) {
        break;
      }
      yield val;
      count += 1;
    }
  }
}
```

Using Modules

We can load exported object using the `import` statement:

```
// js/other.js

import {fibGen, takeN} from "fib.js";

console.log(sum(takeN(fibGen(), 5)));
```

Note

As of now, support for this feature is limited (available behind flag in Chrome 60, Edge 38, Firefox 54 and up). To use imports in non-modern browsers or in node, we need to use Babel to get support:

```
$ npm install --save-dev babel-cli babel-preset-env
```

Promises

Promises are objects that allow for asynchronous programming. They are an alternative for callbacks. If you know that a value might be available in the future, a promise can represent that.

There are three states for a promise:

- Pending - result is not ready
- Fulfilled - result is ready
- Rejected - an error occurred

On the promise object a `then` method will be called to move the state to either fulfilled or rejected.

An async function that implements a promise allows us to *chain* a `then` and a `catch` method:

```
asyncFunction()
  .then(function (result) {
    // handle result
    // Fulfilled
  })
  .catch(function (error) {
    // handle error
    // Rejected
  })
```

Promise Properties

Property	Description
`Promise.length`	Return `1`, number of constructor arguments
`Promise.name`	value: `Promise`
`Promise.prototype`	Prototype for `Promise`

Promise Methods

Method	Description
`Promise.all(promises)`	Return `Promise` that returns when `promises` are fulfilled, or rejects if any of them reject.
`Promise.race(promises)`	Return `Promise` that returns when any of the `promises` reject or fulfill
`Promise.reject(reason)`	Return `Promise` that rejects with `reason`
`Promise.resolve(value)`	Return a `Promise` that fulfills with `value`. If `value` has a `then` method, it will return it's final state

Promise Prototype Methods

Method	Description
`p.p.catch(func)`	Return new `Promise` with rejection handler `func`
`p.p.constructor()`	Return `Promise` function
`p.p.then(fulfillFn, rejectFn)`	Return `Promise` that calls `fulfillFn(value)` on success, or `rejectFn(reason)` on failure

Regular Expressions

A REGULAR EXPRESSION ALLOWS YOU TO MATCH CHARACTERS IN STRINGS. YOU CAN CREATE THEN USING THE LITERAL syntax. Between the slashes place the regular expression. Flags can be specified at the end:

```
let names = 'Paul, George, Ringo, and John'


  let re = /(Ringo|Richard)/g;
```

You can use the `match` method on a string:

```
names.match(re);  //[ 'Ringo', 'Ringo' ]
```

Or the `exec` method on a regex:

```
re.exec(names)  // [ 'Ringo', 'Ringo' ]
```

If there is a match it returns an array. The 0 position is the matched portion, the remaining items correspond to the captured groups. These are specified with parentheses. You can just count the left parentheses (unless they are escaped) in order. Index 1 will be the group of the first parenthesis, index 2 will be the second left parenthesis group, etc.

You can also call the constructor:

```
let re2 = RegExp('Ringo|Richard', 'g')
```

RegExp Flags

FLAG	DESCRIPTION
`g`	Match *globally*, returning all matches, not just first
`i`	*Ignore* case when matching
`m`	Treat newlines as breaks for `^` and `$`, allowing *multi line* matching
`y`	*Sticky* match, start looking from `r.lastIndex` property
`u`	*Unicode* match

Character Classes

CHARACTER	DESCRIPTION
`\d`	Match one digit (`[0-9]`)
`\D`	Match one non-digit (`[^0-9]`)
`\w`	Match one *word* character (`[0-9a-zA-z]`)
`\W`	Match one non-word character (`[^0-9a-zA-z]`)
`\s`	Match one *space* character
`\S`	Match one non-space character
`\b`	Match word boundary
`\B`	Match non-word boundary
`\t`, `\r`, `\v`, `\f`	Match one tab, carriage return, vertical tab, or line feed respectively
`\0`	Match one null character
`\cCHAR`	Match one control character. Where `CHAR` is character
`\xHH`	Match one character with hex code `HH`
`\uHHHH`	Match one UTF-16 character with hex code `HHHH`
`\u{HHHH}`	Match one unicode character with hex code `HHHH` (use `u` flag)

Syntax Characters

CHARACTER	DESCRIPTION
`^`	Match beginning of line (note different in character class)
`$`	Match end of line
`a|b`	Match `a` or `b`
`[abet]`	*Character class* match one of `a`, `b`, `e`, or `t`
`[^abe]`	`^` negates matches in character class. One of not `a`, `b`, or `e`
`\[`	*Escape*. Capture literal `[`. The following need to be escaped `^ $ \ . * + ? ( ) [ ] { } |`
`.`	Match one character except newline
`a?`	Match `a` zero or one time (`a{0,1}`) (note different following `*`, `+`, `?`, `}`)
`a*`	Match `a` zero or more times (`a{0,}`)
`a+`	Match `a` one or more times (`a{1,}`)
`a{3}`	Match `a` three times
`a{3,}`	Match `a` three or more times
`a{3,5}`	Match `a` three to five times
`b.*?n`	`?` match non-greedy. ie from `banana` return `ban` instead of `banan`
`(Paul)`	*Capture* match in group. Captured result will be in position 1 `exec`
`(?:Paul)`	Match `Paul` but don't capture. Result will only be in position 0 in `exec`
`\NUM`	*Backreference* to match previous capture group. `/<(\w+)>(.*)<\/\1>/` will capture an xml tag name in 1 and the content in 2
`Foo(?=script)`	*Assertion* match `Foo` only if followed by `script` (don't capture `script`)
`Foo(?!script)`	*Assertion* match `Foo` only if not followed by `script`

RegExp Properties

Property	Description
`RegExp.length`	value: `2`
`RegExp.name`	value: `RegExp`
`RegExp.prototype`	value: `/(?:)/`
`r.p.flags`	Return flags for regular expression
`r.p.global`	Return boolean if global (`g`) flag is used
`r.p.ignoreCase`	Return boolean if ignore case (`i`) flag is used
`r.p.multiline`	Return boolean if multi line (`m`) flag is used
`r.p.source`	Return string value for regular expression
`r.p.sticky`	Return boolean if sticky (`y`) flag is used
`r.p.unicode`	Return boolean if unicode (`u`) flag is used
`r.sticky`	Return integer specifying where to start looking for next match

RegExp Prototype Methods

Method	Description
`r.p.compile()`	Not useful, just create a `RegExp`
`r.p.constructor()`	Return constructor for `RegExp`
`r.p.exec(s)`	Return array with matches found in string `s`. Index 0 is matched item, other items correspond to capture parentheses. Updates `r` properties
`r.p.test(s)`	Return boolean if `r` matches against string `s`
`r.p.toString()`	Return string representation of `r`

About the Author

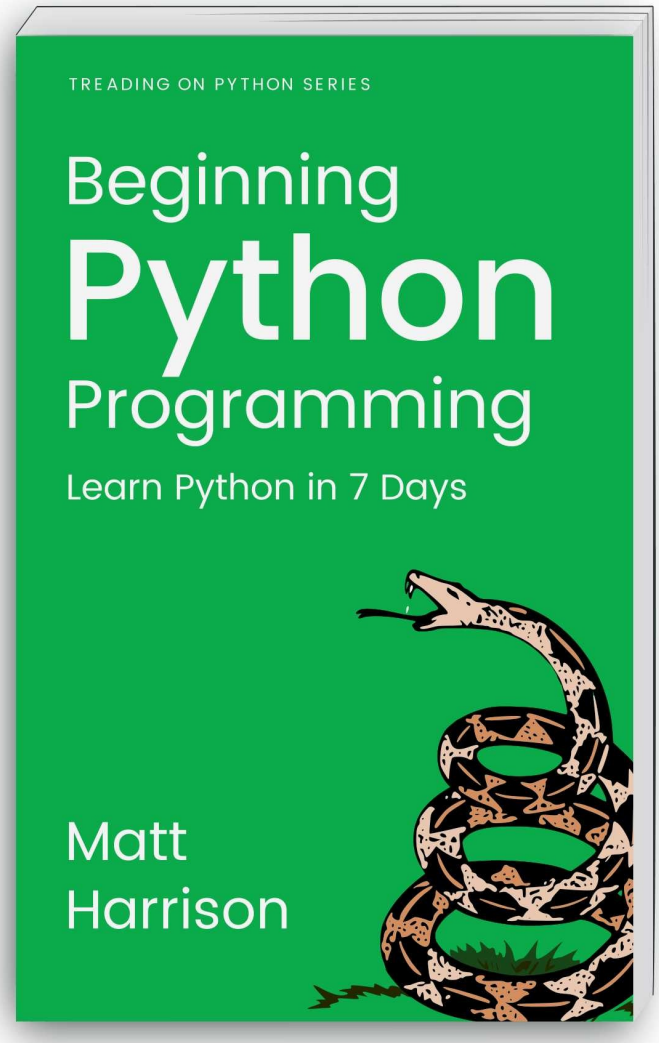
Matt Harrison has been using Python since 2000. He runs MetaSnake, a Python and Data Science consultancy and corporate training shop. In the past, he has worked across the domains of search, build management and testing, business intelligence and storage.

He has presented and taught tutorials at conferences such as Strata, SciPy, SCALE, PyCON and OSCON as well as local user conferences. The structure and content of this book is based off of first hand experience teaching Python to many individuals.

He blogs at `hairysun.com` and occasionally tweets useful Python related information at `@__mharrison__`.

Also Available

Beginning Python Programming



Treading on Python: Beginning Python Programming [2] by Matt Harrison is the complete book to teach you Python fast. Designed to up your Python game by covering the basics:

- Interpreter Usage
- Types
- Sequences
- Dictionaries
- Functions
- Indexing and Slicing
- File Input and Output

- Classes
- Exceptions
- Importing
- Libraries
- Testing
- And more …

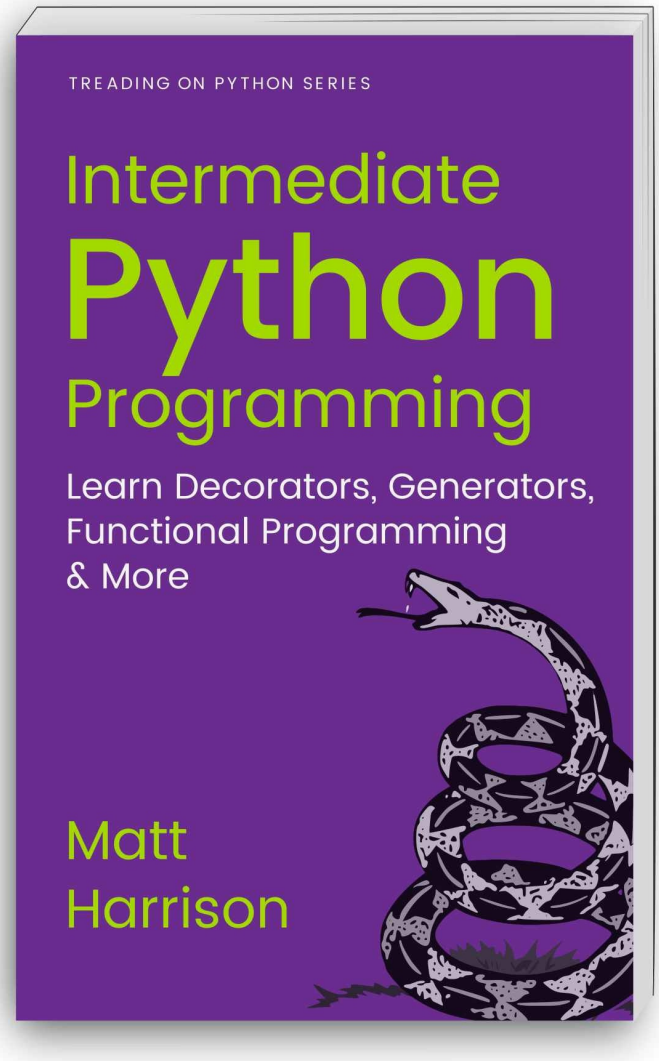
Reviews

Matt Harrison gets it, admits there are undeniable flaws and schisms in Python, and guides you through it in short and to the point examples. I bought both Kindle and paperback editions to always have at the ready for continuing to learn to code in Python.
—S. Oakland

This book was a great intro to Python fundamentals, and was very easy to read. I especially liked all the tips and suggestions scattered throughout to help the reader program Pythonically :)
—W. Dennis

You don't need 1600 pages to learn Python
Last time I was using Python when Lutz's book Learning Python had only 300 pages. For whatever reasons, I have to return to Python right now. I have discovered that the same book today has 1600 pages.
Fortunately, I discovered Harrison's books. I purchased all of them, both Kindle and paper. Just few days later I was on track.
Harrison's presentation is just right. Short and clear. There is no 50 pages about the Zen and philosophy of using if-then-else construct. Just facts.
—A. Customer

Treading on Python: Vol 2: Intermediate Python



Treading on Python: Vol 2: Intermediate Python [3] by Matt Harrison is the complete book on intermediate Python. Designed to up your Python game by covering:

- Functional Programming
- Lambda Expressions
- List Comprehensions
- Generator Comprehensions
- Iterators
- Generators
- Closures
- Decorators
- And more …

Reviews

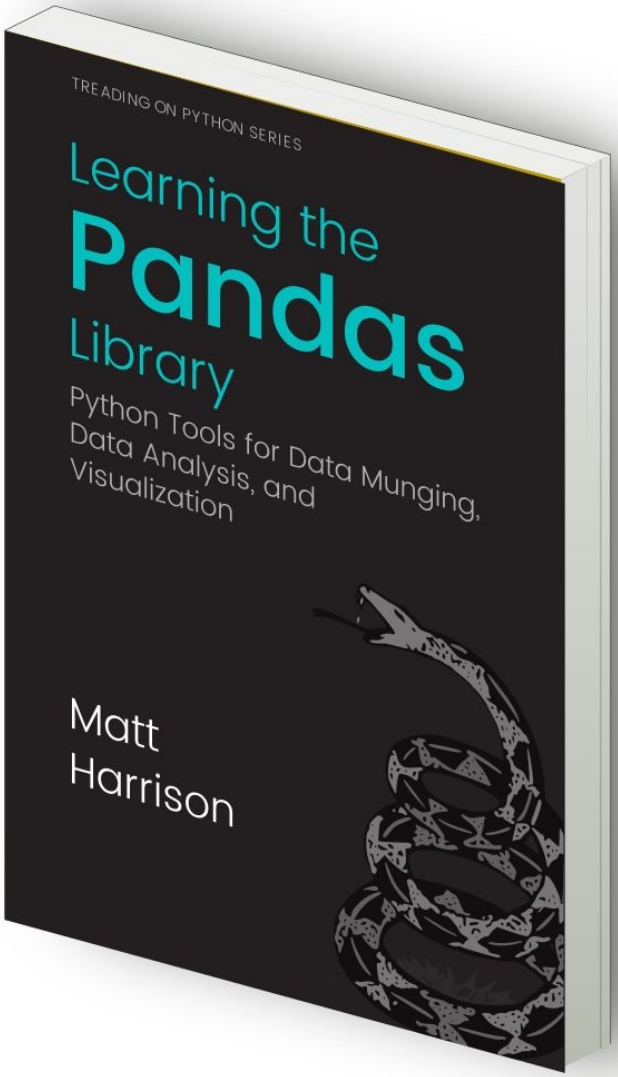
Complete! All you must know about Python Decorators: theory, practice, standard decorators.
All written in a clear and direct way and very affordable price.
Nice to read in Kindle.
—F. De Arruda (Brazil)

This is a very well written piece that delivers. No fluff and right to the point, Matt describes how functions and methods are constructed, then describes the value that decorators offer.
…
Highly recommended, even if you already know decorators, as this is a very good example of how to explain this syntax illusion to others in a way they can grasp.
—J Babbington

Decorators explained the way they SHOULD be explained …
There is an old saying to the effect that "Every stick has two ends, one by which it may be picked up, and one by which it may not." I believe that most explanations of decorators fail because they pick up the stick by the wrong end.
What I like about Matt Harrison's e-book "Guide to: Learning Python Decorators" is that it is structured in the way that I think an introduction to decorators should be structured. It picks up the stick by the proper end…
Which is just as it should be.
—S. Ferg

This book will clear up your confusions about functions even before you start to read about decoration at all. In addition to getting straight about scope, you'll finally get clarity about the difference between arguments and parameters, positional parameters, named parameters, etc. The author concedes that this introductory material is something that some readers will find "pedantic," but reports that many people find it helpful. He's being too modest. The distinctions he draws are essential to moving your programming skills beyond doing a pretty good imitation to real fluency.
—R. Careago

Learning the Pandas Library



Learning the Pandas Library by Matt Harrison is designed to bring developers and aspiring data scientists who are anxious to learn Pandas up to speed quickly. It starts with the fundamentals of the data structures. Then, it covers the essential functionality. It includes many examples, graphics, code samples, and plots from real world examples.

- Installation
- Data Structures
- Series CRUD
- Series Indexing
- Series Methods
- Series Plotting
- Series Examples
- DataFrame Methods

- DataFrame Statistics
- Grouping, Pivoting, and Reshaping
- Dealing with Missing Data
- Joining DataFrames
- DataFrame Examples

Reviews

This is an excellent introduction benefitting from clear writing and simple examples. The pandas documentation itself is large and sometimes assumes too much knowledge, in my opinion. Learning the Pandas Library bridges this gap for new users and even for those with some pandas experience such as me.
—Garry C.

I have finished reading Learning the Pandas Library and I liked it... very useful and helpful tips even for people who use pandas regularly.
—Tom. Z

Ebook Formatting: KF8, Mobi & EPUB

Ebook Formatting: KF8, Mobi & EPUB by Matt Harrison is the complete book on formatting for all Kindle and EPUB devices. A good deal of the eBooks found in online stores today have problematic formatting. *Ebook Formatting: KF8, Mobi & EPUB* is meant to be an aid in resolving those issues. Customers hate poorly formatted books. Read through the reviews on Amazon and you'll find many examples. This doesn't just happen to self-published books. Books coming out of big Publishing Houses often suffer similar issues. In the rush to put out a book, one of the most important aspects affecting readability is ignored.

This books covers all aspects of ebook formatting on the most popular devices (Kindle, iPad, Android Tablets, Nook and Kobo):

- Covers
- Title Pages
- Images
- Centering stuff
- Box model support

- Text Styles
- Headings
- Page breaks
- Tables
- Blockquotes
- First sentence styling
- Non-ASCII characters
- Footnotes
- Sidebars
- Media Queries
- Embedded Fonts
- Javascript
- and more …

2 - http://hairysun.com/books/tread/

3 - http://hairysun.com/books/treadvol2/

One more thing

THANK YOU FOR BUYING AND READING THIS BOOK.

If you have found this book helpful, I have a big favor to ask. As a self-published author, I don't have a big Publishing House with lots of marketing power pushing my book. I also try to price my books so that they are much more affordable.

If you enjoyed this book, I hope that you would take a moment to leave an honest review on Amazon. A short comment on how the book helped you and what your learned makes a huge difference. A quick review is useful to others who might be interested in the book.

Thanks again!