

2018 SpeedStudy ¹

P1xt

January 4, 2018

¹An adventure to explore how much of MITOCW's (Computer Science, Physics and Mathematics) Curriculum I can thoroughly learn in a year.

ABSTRACT

I'm tackling my 2018 learning goals in true "Speedrun" style. The following list contains far more than I expect I'll be able to complete in 2018, so it's sufficient that I can speedrun the year without running out of items from the list to learn.

My dual main goals are to A) study algorithms in depth and B) explore Physics. To that end, I am including a variety of Computer Science, Mathematics and Physics courses, with no prescribed order. I'll just select the next course in whatever list I'm most inspired to continue each time I'm ready to progress to a new course.

Additionally, I'm including a variety of "bonus point" opportunities for reading books, completing projects, and completing algorithmic challenges.

My goal is to see how many points I can rack up by December 31, 2018.

Current Course: 18.01

Total Points: 0

Activity	Points
Basic Project	100
Substantial Project	200
Large Project	300
Gigantic Project	400
Duolingo (1 level)	100
Book	200
Tutorial site (Udemy/Pluralsight/Egghead/Treehouse- /etc) course	100
University level course (Coursera/edX/MITOCW/etc)	500
Physical Activity (30 minutes)	10
Analytics Vidhya Competition	50
Crowd Analytix Competition	50
Kaggle Competition	50
Driven Data Competition	50
Blog Post on any topic	15
Blog Post Tutorial	40
Video Tutorial	50
Open Source PR	50
Module to npm	200
Team Up for a project	100
CodeWars (10 problems)	50
CodinGame (1 Tier)	100
CodinGame (1 Bot Competition)	20
HackerRank (10 problems)	50
Google Code Jam (1 round from past contest)	100

CONTENTS

Abstract	2
Course: 6.0001	4
Notes	4
Problem Set 0	6
Problem Set 1	6
Problem Set 2	6
Problem Set 3	6
Problem Set 4	6
Problem Set 5	6

COURSE: 6.0001

NOTES

All knowledge falls into two categories - **Declarative knowledge** (statements of fact) and **Imperative knowledge** (instructions "how to" compute something).

An **Algorithm** is a list of instructions that describe a computation that when executed on a set of inputs will proceed through a set of well-defined states and eventually produce an output.

Fixed-program computers only have the code necessary to do one thing, whereas **stored program computers** are more flexible and execute any computation in their instruction set.

Syntax - which strings of characters and symbols are well formed.

Semantics - associates a meaning with each syntactically correct string that has no static semantic errors.

Static Semantics - which syntactically valid strings have meaning.

Scalar objects are indivisible, whereas **non-scalar objects** such as strings have internal structure. Python's four types of scalar objects are int, float, bool, and None.

Objects and Operators are combined to form **Expressions** which each evaluate to an object of some type.

Operations on strings, tuples, ranges and lists - `seq[i]`, `len(seq)`, `seq1+seq2`, `n*seq`, `seq[start:end]`, `e in seq`, `e not in seq`, for `e in seq`.

Strings - can be indexed into, sliced, and length can be found with the `len` function. Contain only characters. Methods include: `s.count(s1)`, `s.find(s1)`, `s.rfind(s1)`, `s.index(s1)`, `s.rindex(s1)`, `s.lower()`, `s.replace(old, new)`, `s.rstrip()`, `s.split(d)`.

Tuples - immutable ordered sequences of elements, like strings, but can be ordered sequences of any type, not just characters. Format - ('a', 'b', 1, 1.2, 'a'). Can be concatenated, indexed and sliced like strings.

Ranges - immutable like strings and tuples but contain only integers. The `range` function returns

an object of type `range` and takes three arguments, start, stop and step. Step may be negative. Step default is 1. Start defaults to 0.

Lists - like tuples and can contain collections composed of any type of object, but use square brackets and are mutable. Use `id()` to get the id of a list object. Can use `append(e)`, `count(e)`, `insert(i,e)`, `extend(L1)`, `remove(e)`, `index(e)`, `pop(i)`, `sort()`, `reverse()`.

List Comprehension - creates a new list in which each element is the result of applying a given operation to a value from a sequence (another list). Like `.map()` in JavaScript. Syntax: `L = [x**2 for x in range(1, 7)]` will cause L to contain [1, 4, 9, 16, 25, 36].

Dictionaries - objects of type `dict`, like lists but are indexed as key/value pairs. Like lists, are mutable. Use keys that are **hashable types** which include all immutable data types, but no mutable ones. Methods available on dicts: `len(d)`, `d.keys()`, `d.values()`, `k in d`, `d[k]`, `d.get(k, v)`, `d[k]=v`, `del d[k]`, for `k in d`.

Classes - used to implement data abstractions, define an interface between the data type and the rest of the program, comprised of both objects and available operations on those objects. Can be instantiated to create a new instance of the class, and uses dot notation to access attributes associated with the class. Special methods `__init__` and `__str__` initialize an object of the class, and return a string representation of the object. **Inheritance** allows an class of one type to inherit attributes from another class. Subclasses extend the behavior of their superclass. The Subclass should respond properly to any request that it's superclass could handle, the converse is not necessarily true.

Encapsulation - bundling together data attributes and the methods for operating on them.

Information Hiding - how methods in a class are implemented are irrelevant to clients of that class so long as they conform to the specification they agree to.

Generators - any function containing a `yield` statement. Typically used in conjunction with `for` statements.

Lambda expression - anonymous function, syntax: `lambda <variables>: expression`

Floats - are approximations of values containing a fractional or decimal amount. May not be exact. Significant digits denote the precision. An approximation is the rounded value.

Type Casting allows for conversion between types. You can convert one type to another by using the name of the result type you want. `int('23')` converts the string 23 to the integer 23.

Conditionals allow a program to branch depending on some condition (if x do y, else do z)

Iteration allows a program to repeat some series of instructions until a condition is met.

Computational Complexity is the study of the intrinsic difficulty of problems. Typically uses the asymptotic notation "Big O" which is used to give an upper bound on the order of growth of a function.

Constant time implies that the running time of a program does not increase with the size of the input to the program.

Bisection search - approximates by making educated guess at high and low values, checking the midpoint between the two, and then replacing either the high or low value with the mid value and repeating until a "close enough" value is found. Effectively halves the search domain each iteration.

Successive approximation - Newton-Raphston method for finding the square root

$$guess - \frac{p(guess)}{p'(guess)}$$

is a better approximation for the root of a polynomial than *guess*.

Recursion - when a function calls itself to iteratively solve a smaller subset of the same problem.

Modules - a .py file containing Python definitions and statements. A program may consist of many modules.

Files - can be accessed via `open(fh, 'w| r| a')`, `fh.read()`, `fh.readline()`, `fh.readlines()`, `fh.write(s)`, `fh.writeLines(S)`, `fh.close()`

Testing - running a program to determine whether it works as intended. The intention is to show what bugs exist. **Glass box** tests paths through a program whereas **Black box** tests paths through the specification. Unit testing tests individual functions work properly, integration testing tests the program as a whole. Regression testing ensures that what used to work still works. Drivers

simulate part of the program that use the unit being tested, stubs simulate parts of the program used by the unit being tested.

Exception - something that does not conform to the norm. Should be handled by the program. Use `try: except <exception>: block to handle`.

Assertions - confirm that the state of a computation is as expected. Will raise an `AssertionError` exception if it evaluates to False.

Debugging - attempting to fix a program that you know isn't working properly.

`print()` - prints to console

`input('prompt string')` - gets input from the user

`type(X)` - returns the type of X

`while <condition>:` - loops through indented statements while the condition is true

`for <item in list>:` - loops through each item, use `break` to exit the loop early

`def <functionName(x, y)>:` - defines a function, `return` exits the function, can pass parameters by position or keyword (like `y='something'` as a parameter), can set default in parameter list, like `y='something'` to be used when no y is passed.

Higher-order functions take another function as an argument.

Read everything through Chapter 10. Currently at 8.2.1

PROBLEM SET 0

Problem .

Solution.



PROBLEM SET 1

Problem .

Solution.



PROBLEM SET 2

Problem .

Solution.



PROBLEM SET 3

Problem .

Solution.



PROBLEM SET 4

Problem .

Solution.



PROBLEM SET 5

Problem .

Solution.

