CS50 - SECTION 6

10/23/18

POST-MORTEM
ON
PSET 5

QUICK REMINDERS

1. Avoid redundancy by refactoring your styles appropriately.

```
<p style="font-color: purple; font-size: 15px;">Text 1</p>
<p style="font-color: green; font-size: 15px;">Text 2</p>
<p style="font-color: orange; font-size: 15px;">Text 3</p>
```

QUICK REMINDERS

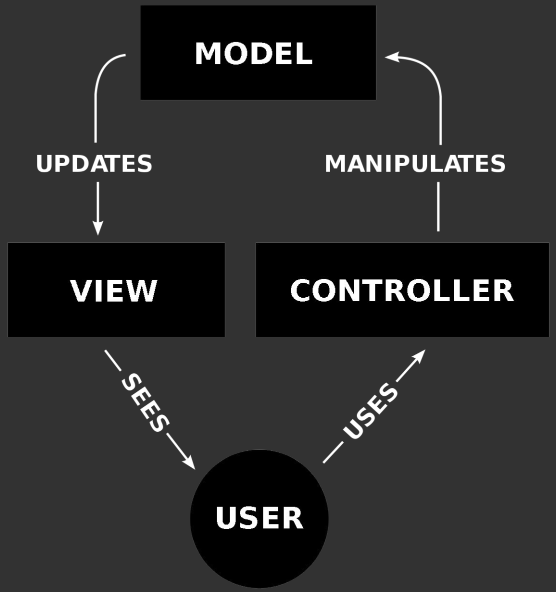
1. Avoid redundancy by refactoring your styles appropriately.

```
<p style="font-color: purple; font-size: 15px;">Text 1</p>
<p style="font-color: green; font-size: 15px;">Text 2</p>
<p style="font-color: orange; font-size: 15px;">Text 3</p>
```

The `font-size` property is repeated and could be refactored into a CSS class property

QUICK REMINDERS

2. Don’t include large amounts of styling in your HTML.



QUICK REMINDERS

3. You can only have one ID per element. Similarly, you can’t have duplicate key-value pairs for your attributes in HTML.

```
<p id="para item">Text</p>
```

QUICK REMINDERS

3. You can only have one ID per element. Similarly, you can’t have duplicate key-value pairs for your attributes in HTML.

~~`<p id="para item">Text</p>`~~

```
<p id="para">Text</p>
```

QUESTIONS?

Any questions about HTML, CSS, and/or JavaScript before we move forward?

CONCEPTS DEEP-DIVE

"SHORTS" FOR THE WEEK

https://youtu.be/8xCzjOnfQbw

https://youtu.be/jOKx1JkRlho

INTRODUCING PYTHON

- Python is a **dynamically typed** (types determined at runtime) and **strongly typed** (you can’t mix types, such as trying to add an integer and string together) language
  - C is **statically typed** (types explicitly defined by you) and strongly typed
- Python is an **interpreted language** (instructions run line-by-line without compiling)
  - C was a **compiled language** (compiled into object files before you can run it)

INTRODUCING PYTHON

- What are the advantages/disadvantages of an interpreted language?

IMPORTANT CAVEAT

VARIABLES

- Variables in Python *do* have underlying types, despite the fact that we don't have to explicitly declare them

```
coursenum = 50
coursename = "Introduction to Computer Science I"
```

- When we reassign a variable in Python, it's like we "rip off the label" and point to another container (meaning types can change)

VARIABLE TYPES

- We have a number of types we can choose from:
  - number
  - string
  - tuple
  - list
  - dictionary
  - set

VARIABLE TYPES

- We have a number of types we can choose from:
  - number
  - string
  - tuple
  - list
  - dictionary
  - set

Notice that we don't have to specify between `int`, `double`, `float`, etc.

VARIABLE TYPES

- We have a number of types we can choose from:
  - number
  - string
  - tuple
  - list
  - dictionary
  - set

Notice that we don't have to specify between `int`, `double`, `float`, etc.

Notice that we don't have an array of characters anymore!

TUPLES

- The most basic data structure in Python

```
t = (1, 2, "apple", 4.5)
```

- *Can* contain elements of different types
- Is immutable—You cannot increase/decrease it’s size
  - Perhaps closest to an array in C
- Notice we use parentheses to create them

TUPLES

- Tuples allow for **unpacking**, in which you split up their values and assign them to different variables:

```
coordinate = (3, 2, 7)
 x, y, z = coordinate
```

LISTS

- Similar to tuples, but are *not* immutable

```
l = [1, 2, "apple", 4.5]
```

- *Can* contain elements of different types
- You can increase/decrease their size as you wish
- Notice we use square brackets to create them

WORKING WITH LISTS

OPERATION	DESCRIPTION
`list.append(x)`	Appends an item **`x`**
`list.extend(x,y,z)`	Extends a list with items **`x`**, **`y`**, **`z`**
`list.insert(i, x)`	Inserts **`x`** at position **`i`**
`list.remove(x)`	Removes first item in the list whose value is equal to **`x`**
`del list[i:k]`	Removes elements from **`i`** to **`k`**

SETS

- Sets are unordered and contain no duplicate elements

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

- *Can* contain elements of different types
- You can increase/decrease their size as you wish
- Notice we use curly brackets to create them

WORKING WITH SETS

OPERATION	DESCRIPTION
`set.union(s)`	Returns all elements in `set` and/or `s`
`set.intersection(s)`	Returns all elements common to `set` and `s`
`set.difference(s)`	Returns all elements in `set`, but not `s`
`set.add(x)`	Adds element `x` to the set
`set.remove(x)`	Removes element `x` from the set

DICTIONARIES

- Dictionaries follow the key-value pair structure

```
tel = {'jack': 4098, 'sape': 4139}
```

- *Can* contain elements of different types
- You can increase/decrease their size as you wish
- Notice we use curly brackets to create them
- You access a specific value by indicating the key it belongs to: `tel['jack']` returns `4098`

WORKING WITH DICTIONARIES

OPERATION	DESCRIPTION
`dict[“existing_key”] = <val>`	Sets **`existing_key`** to **`<val>`**
`dict[“new_key”] = <val>`	Adds **`new_key`** to dictionary and sets it equal to **`<val>`**
`del dict[“existing_key”]`	Deletes the key-value pair for **`existing_key`** from the dictionary

FINDING THE LENGTH OF DATA STRUCTURES

- You can use **`len(x)`** to find the length of any data structure in Python

INTERACTIVE DEMO OF DATA STRUCTURES

http://bit.ly/2RbGQGB

CONDITIONALS

C

```c
int x = get_int();
if (x < 0)
{
   printf("x is negative\n");
}
else if (x > 0)
{
   printf("x is positive\n");
}
else
{
   printf("x is zero\n");
}
```

PYTHON

```python
x = cs50.get_int()
if x < 0:
   print("x is negative")
elif x > 0:
   print("x is positive")
else:
   print("x is zero")
```

SOME KEY DIFFERENCES WITH CONDITIONALS

- We use the keywords **`and`**, **`or`**, **`not`** instead of **`&&`**, **`||`**, **`!`**
- Use **`elif`** instead of **`else if`**
- No equivalent of the **`switch`** statement in Python
- Body code introduced with a **`:`** instead of **`{}`**
  - **Must be indented and whitespace matters!**

THE WHILE LOOP

C

```c
while(i < 100)
{
    printf("%i\n", ++i);
}
```

PYTHON

```python
i = 0
while i < 100:
    print(i)
    i += 1
```

THE WHILE LOOP

C

```
while(i < 100)
{
    printf("%i\n", ++i);
}
```

PYTHON

```
i = 0
while i < 100:
    print(i)
    i += 1
```

Notice the use of indentation and the : symbol

THE FOR LOOP

C

```c
for(int j = 0; j < 100; j += 2)
{
   printf("%i\n", j);
}
```

PYTHON

```python
for j in range(0, 101, 2):
   print(j)
```

THE FOR LOOP

- The **`while`** loop is quite similar to its C counterpart, but the **`for`** loop is much more robust and powerful in Python than in C
- **`for`** loops in Python don't actually iterate over indices, but instead iterate over sequences

```
for j in range(0, 101, 2):
   print(j)
```

`range()` returns a sequence from 0 to 101, counting up by 2 each time:

(0, 2, 4, ..., 100)

FUNCTIONS

- Just like in C, we have functions which have an input and output
- However, functions are modified to fit the “Pythonic” style:
  - You don’t have to specify types for the parameter list
  - You don’t have to specify a return type (including for **`void`** functions)
  - Introduce functions with the **`def`** keyword

FUNCTIONS

```
def square(x):
    return x ** 2

base = cs50.get_float()
print(square(base))
```

FUNCTIONS

```
def square(x):
    return x ** 2


base = cs50.get_float()
print(square(base))
```

Indentation and whitespace matters! ***Are you catching onto a theme?***

FUNCTIONS

```
def square(x):
    return x ** 2


base = cs50.get_float()
print(square(base))
```

Indentation and whitespace matters! ***Are you catching onto a theme?***

Notice because we have simplified types (just `number`), we don't have to handle different number types individually

OBJECTS

- In C, we could create our own new “data types” by establishing structures:

```
struct address
{
  char name[50];
  char street[100];
  char city[50];
  char state[20];
  int pin;
};
```

OBJECTS

- Python has this functionality as well, but through **objects**
- Objects in Python can have properties and methods
  - Just like in C, **methods** are the fields of data we want to store
  - However, new to Python, we can also give objects **methods** which are functions inherently part of that object
- We used a **struct** to define the "template" of a structure in C, and we use a **class** to define an object's "template" in Python

OBJECTS

```python
class Student():
    def __init__(self, name, year="Freshman"):
        self.name = name
        self.year = year

    def endYear(self):
        if self.year == "Freshman":
            self.year = "Sophomore"
        elif self.year == "Sophomore":
            self.year = "Junior"
        elif self.year == "Junior":
            self.year = "Senior"
        else:
            self.year = "Alum"

    def info(self):
        print(f"{self.name} is a {self.year}.")
```

OBJECTS

```python
class Student():
    def __init__(self, name, year="Freshman"):
        self.name = name
        self.year = year

    def endYear(self):
        if self.year == "Freshman":
            self.year = "Sophomore"
        elif self.year == "Sophomore":
            self.year = "Junior"
        elif self.year == "Junior":
            self.year = "Senior"
        else:
            self.year = "Alum"

    def info(self):
        print(f"{self.name} is a {self.year}.")
```

Use the `class` keyword to define a new class

The **__init__** function is our object constructor

We can include as many functions as we want to define

OBJECTS

```
# create two new students, one is a freshman
maria = Student("Maria", "Senior")
newkid = Student("John Harvard")

# everyone graduates at the end of the year
maria.endYear()
newkid.endYear()

# new years, now!
maria.info()
newkid.info()
```

Declaring new objects

Calling the methods of an object

OBJECTS

- When we create a struct in C, we can declare it empty (just reserving space for it in memory) or initialize it with values
  - Similarly, we can create a new *instance* of an object in Python with or without initialized values
  - Python looks for **`__init__`** in our class definition and starts there <u>when you declare a new object</u>

OBJECTS

- We use the `self` keyword so that our methods can operate on our object
  - You are effectively “passing” the object to its own methods to perform some set of operations on it
- You don’t need to use `self` when calling the methods of an object; This is done automatically for you
  - But `self` is always needed as the first parameter of every method you define in the class (if you want your methods to do something to the object itself)

HANDS ON PRACTICE

http://bit.ly/2RcgIeF

HANDS ON PRACTICE - SOLUTIONS

http://bit.ly/2R4NDSa

PROBLEM SET 6 PREVIEW

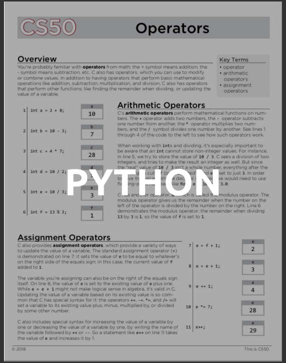
PROBLEM SET 6 PREVIEW

Implement the following:

- `hello.py`
- `mario.py [less]` **OR** `mario.py [more]`
- `cash.py` **OR** `credit.py`
- `caesar.py` **OR** `crack.py` **OR** `vigenere.py`
- `bleep.py`

Be critical of your old C code as you port it to Python. Review the comments on your code, think about the design flaws you made, and optimize your algorithms in Python.

REFERENCE SHEETS



CS50 Operators

Overview

You're probably familiar with **operators** from math: the + symbol means addition, the - symbol means subtraction, etc. C also has operators, which you can use to modify or combine values. In addition to having operators that perform basic mathematical operations like addition, subtraction, multiplication, and division, C also has operators that perform other functions: like finding the remainder when dividing, or updating the value of a variable.

Key Terms

- operator
- arithmetic operators
- assignment operators

```
1  int a = 2 + 8;
2  int b = 10 - 3;
3  int c = 4 * 7;
4  int d = 10 / 2;
5  int e = 10 / 3;
6  int f = 13 % 3;
```

a	b	c	d	e	f
10	7	28	5	3	1

Arithmetic Operators

C's **arithmetic operators** perform mathematical functions on numbers. The + operator adds two numbers, the - operator subtracts one number from another, the * operator multiplies two numbers, and the / symbol divides one number by another. See lines 1 through 4 of the code to the left to see how such operators work.

When working with ints and dividing, it's especially important to be aware that an int cannot store non-integer values. For instance, in line 5, we try to store the value of 10 / 3. C sees a division of two integers, and tries to make the result an integer as well. But since the "real" value of 10 / 3 isn't a whole number, everything after the decimal gets cut off (or "truncated") and e is set to just 3. In order to save the value with the decimal included, we would need to use floating-point numbers like float e = 10.0 / 3.0.

C has another operator, %, which is called the modulus operator. The modulus operator gives us the remainder when the number on the left of the operator is divided by the number on the right. Line 6 demonstrates the modulus operator: the remainder when dividing 13 by 3 is 1, so the value of f is set to 1.

Assignment Operators

C also provides **assignment operators**, which provide a variety of ways to update the value of a variable. The standard assignment operator (=) is demonstrated on line 7: it sets the value of e to be equal to whatever's on the right side of the equals sign: in this case, the current value of f added to 1.

The variable you're assigning can also be on the right of the equals sign itself. On line 8, the value of e is set to the existing value of e plus one. While e = e + 1 might not make logical sense in algebra, it's valid in C. Updating the value of a variable based on its existing value is so common that C has special syntax for it: the operators +=, -=, *=, and /= will set a variable to its existing value plus, minus, multiplied by, or divided by some other number.

C also includes special syntax for increasing the value of a variable by one or decreasing the value of a variable by one, by writing the name of the variable followed by ++ or --. So a statement like e++ on line 11 takes the value of e and increases it by 1.

```
7   e = f + 1;
8   e = e + 1;
9   e += 1;
10  e *= 7;
11  e++;
```

e	e	e	e	e
2	3	4	28	29

© 2018

This is CS50.

PYTHON

https://www.dropbox.com/sh/5y662ey1hc4sde4/AAB-m8F3-J_9P1i2Bng21fgua/Python.pdf?dl=0

CS50 Operators

Overview

You're probably familiar with operators from math: the + symbol means addition, the - symbol means subtraction, etc. C also has operators, which you can use to modify or combine values. In addition to having operators that perform basic mathematical operations like addition, subtraction, multiplication, and division, C also has operators that perform other functions: like finding the remainder when dividing, or updating the value of a variable.

Key Terms: operator; arithmetic operators; assignment operators

1 int a = 2 + 8; 2 int b = 10 - 3; 3 int c = 4 * 7; 4 int d = 10 / 2; 5 int e = 10 / 3; 6 int f = 13 % 3;

a 10; b 7; c 28; d 5; e 3; f 1

Arithmetic Operators

C's arithmetic operators perform mathematical functions on numbers. The + operator adds two numbers, the - operator subtracts one number from another, the * operator multiplies two numbers, and the / symbol divides one number by another. See lines 1 through 4 of the code to the left to see how such operators work.

When working with ints and dividing, it's especially important to be aware that an int cannot store non-integer values. For instance, in line 5, we try to store the value of 10 / 3. C sees a division of two integers, and tries to make the result an integer as well. But since the "real" value of 10 / 3 isn't a whole number, everything after the decimal gets cut off (or "truncated") and e is set to just 3. In order to save the value with the decimal included, we would need to use floating-point numbers like float e = 10.0 / 3.0.

C has another operator, %, which is called the modulus operator. The modulus operator gives us the remainder when the number on the left of the operator is divided by the number on the right. Line 6 demonstrates the modulus operator: the remainder when dividing 13 by 3 is 1, so the value of f is set to 1.

Assignment Operators

C also provides assignment operators, which provide a variety of ways to update the value of a variable. The standard assignment operator (=) is demonstrated on line 7: it sets the value of e to be equal to whatever's on the right side of the equals sign: in this case, the current value of f added to 1.

The variable you're assigning can also be on the right of the equals sign itself. On line 8, the value of e is set to the existing value of e plus one. While e = e + 1 might not make logical sense in algebra, it's valid in C. Updating the value of a variable based on its existing value is so common that C has special syntax for it: the operators +=, -=, *=, and /= will set a variable to its existing value plus, minus, multiplied by, or divided by some other number.

C also includes special syntax for increasing the value of a variable by one or decreasing the value of a variable by one, by writing the name of the variable followed by ++ or --. So a statement like e++ on line 11 takes the value of e and increases it by 1.

7 e = f + 1; 8 e = e + 1; 9 e += 1; 10 e *= 7; 11 e++;

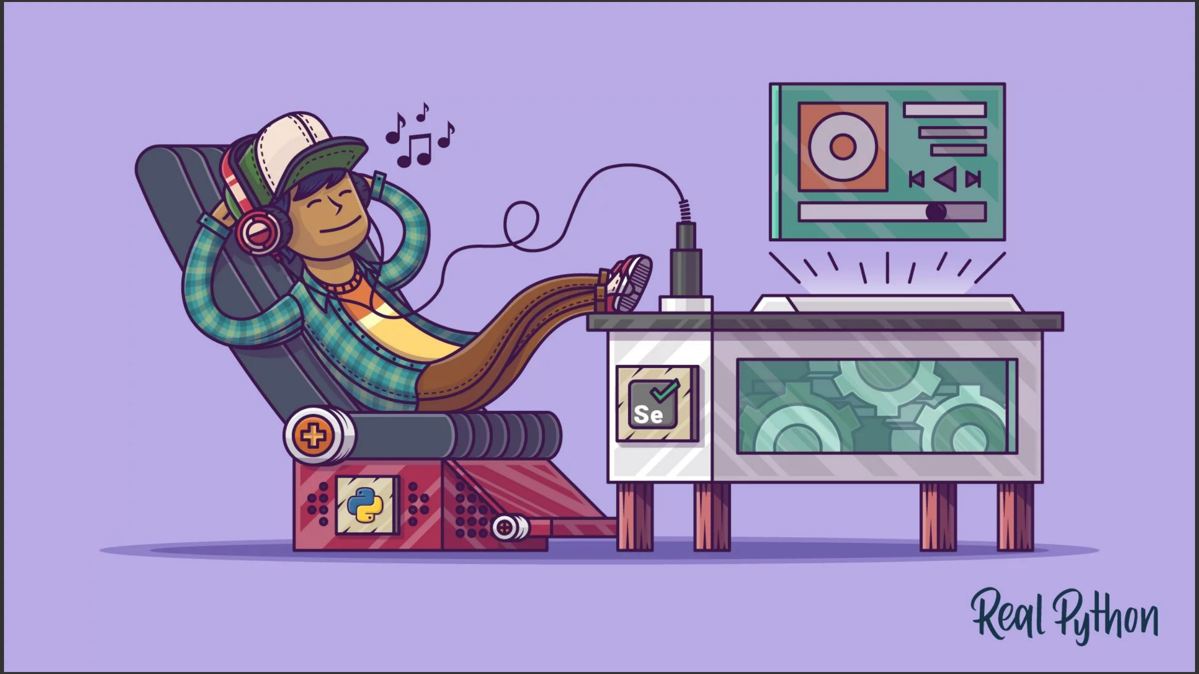
e 2; e 3; e 4; e 28; e 29

© 2018 This is CS50.

PYTHON FOR WEB PROGRAMMING

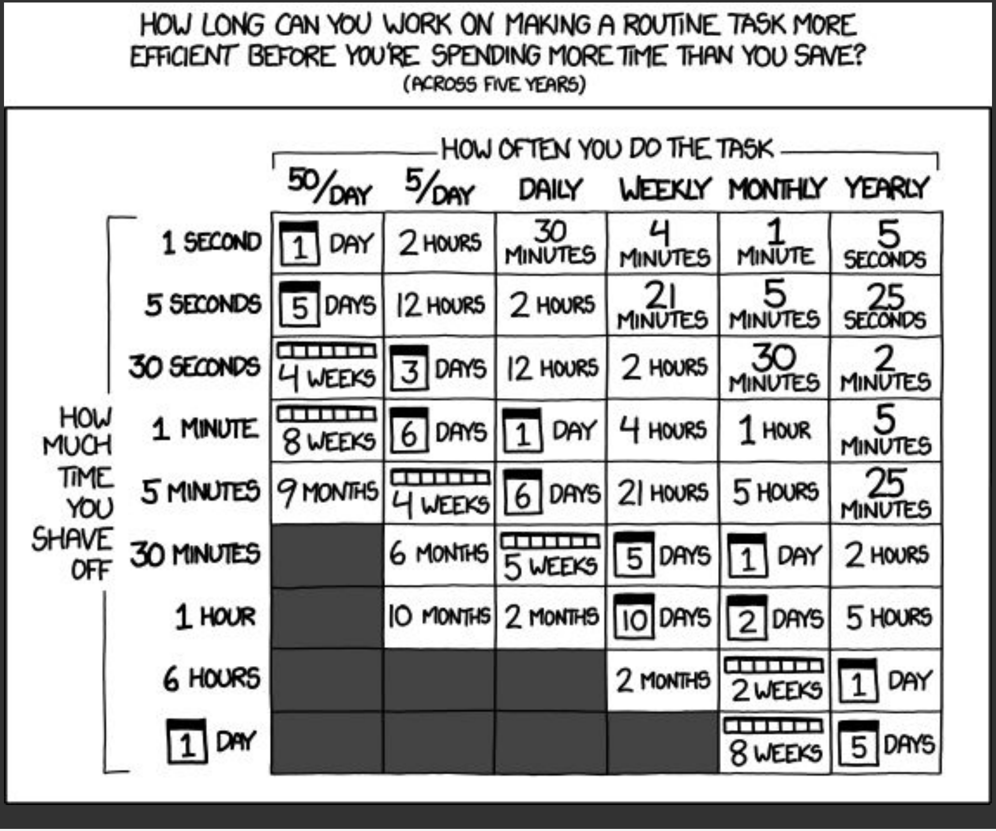
https://www.dropbox.com/sh/5y662ey1hc4sde4/AACyi40IXbh_tF72FrIzyQmfa/Python%20for%20Web%20Programming.pdf?dl=0

PYTHON FOR AUTOMATION



Python is great for automation!

PYTHON FOR AUTOMATION



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

	HOW OFTEN YOU DO THE TASK					
HOW MUCH TIME YOU SHAVE OFF	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES	▨	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR	▨	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS	▨	▨	▨	2 MONTHS	2 WEEKS	1 DAY
1 DAY	▨	▨	▨	▨	8 WEEKS	5 DAYS