HEROES Academy Computer Science Cookbooks Documentation

Release 1.0.0

Brian McMahan

Mar 06, 2017

Contents

1 Python Essentials 3

2 Tutorials 25

Welcome to the HEROES Cookbook. For now, there is only the essentials of Python.

You should use these examples as part of your problem solving. You should ask yourself “what is it that I need to do?” and then find an example that does what you need to do. Using that example as a guide, you can then write code that does what you want!

Note: If you are using Python 2, there are a couple built-in commands that change when using Python 3. But, there is some (coming soon) code to make it act like Python 3.

CHAPTER 1

Python Essentials

Primitive Variables

Numbers

Integers

```
# create an integer
x = 5

# convert an integer string
x = str('5')

# convert a float to an integer
## note: don't depend on this for rounding, it rounds in weird ways
x = int(5.5)

# convert a string of any number base
# for example, binary
x = int('1010101', base=2)
```

Floats

```
# create a float
x = 5.5

# convert a float string
x = float("5.5")

# convert an integer to a float
x = float(5)
```

Basic math operations

```python
x = 100

# 1. Add
x = x + 5
x += 5

# 2. Subtract
x = x - 5
x -= 5

# 3. Multiply
x = x * 5
x *= 5

# 4. Divide
x = x / 5
x /= 5

# 5. Power
x = x ** 2
x **= 2
```

Advanced math operations

```python
# 1. Integer Division
x = x // 5
x //= 5

# 2. Modulo
x = 84
x = x % 5
x %= 5
```

Use the math library

```python
import math

x = 10

# pow is power, same as x ** 2
x = math.pow(x, 2)

# ceil rounds up and floor rounds down
x = 5.5
y = math.ceil(x) # y is 6.0
z = math.floor(x) # z in 5.0

# some other useful ones:
math.sqrt(x)
math.cos(x)
math.sin(x)
math.tan(x)
```

```

# this will give you pi:
math.pi
```

Strings

Add two strings together

```
first_name = "euclid "
space = " "
last_name = "von rabbitstein"
full_name = first_name + space + last_name
```

Repeat a string

```
message = "Repeat me!"
repeated10 = message * 10

# I like to use it for pretty printing code results
line = "-" * 12
print("   Title!   ")
print(line)
```

Index into a string

```
first_name = "Euclid"
last_name = "Von Rabbitstein"
first_initial = first_name[0]
last_initial = last_name[0]
initials = first_initial + last_initial
```

Slice a string

```
# the syntax is
#   my_string[start:stop]
# this includes the start position but goes UP TO the stop
# you can leave either empty to go to the front or end

target = "door"
last_three = target[1:]
first_three = target[:3]
middle_two = target[1:3]

# you can use negatives to slice off the end!
all_but_last = target[:-1]

pig_latin = target[1:] + target[0] + "ay"
```

String's inner functions

```python
full_name = "euclid von Rabbitstein"

# all caps
full_name_uppered = full_name.upper()

# all lower
full_name_lowered = full_name.lower()

# use lower to make sure something is lower before you compare it
user_command = "Exit"
if user_command.lower() == "exit":
    print("now I can exit!")

# first letter capitalized
full_name_capitalized = full_name.capitalize()

# split into a list
full_name_list = full_name.split(" ")

# strip off any extra spaces
test_string = "   extra spaces everywhere   "
stripped_string = test_string.strip()

# replace things in a string
full_name_replaced = full_name.replace("von", "rabbiticus")

# use replace to delete things from a string!
test_string = "annoying \t tabs in \t the string"
fixed_string = test_string.replace("\t","")
```

Boolean algebra

Create a literal boolean variable

```python
literal_boolean = True
other_one = False
```

Create a boolean variable from comparisons

```python
x = 9
y = 3
x_is_bigger = x > y # True
x_is_even = x % 2 == 0 # False
x_is_multiple_of_y = x % y == 0 # True
```

Combine two boolean variables with 'and' and 'or'

```
# example data
card_suit = "Hearts"
card_number = 7

# save the results from comparisons!
card_is_hearts = card_suit == "Hearts"
card_is_diamond = card_suit == "Diamond"
card_is_big = card_number > 8

# only 1 of them needs to be true
card_is_red = card_is_hearts or card_is_diamond

# both need to be true
card_is_good = card_is_red and card_is_big

# creates the opposite!
card_is_bad = not card_is_good
```

Containers

Lists

Create an empty list

```
new_list = list()
# or
new_list = []
```

Create a list with items

```
my_pets = ['euclid', 'leta']
```

Add onto a list

```
my_pets.append('socrates')
```

Index into a list

```
first_pet = my_pets[0]
second_pet = my_pets[1]
third_pet = my_pets[2]
```

Slice a list into a new list

```python
# the syntax is
#   my_list[start:stop]
# this includes the start position but goes UP TO the stop
# you can leave either empty to go to the front or end

first_two_pets = my_pets[:2]
last_two_pets = my_pets[1:]
```

Test if a value is inside a list

```python
## with any collection, you can test if an item is inside the collection
## it is with the "in" keyword

my_pets = ['euclid', 'leta']
if 'euclid' in my_pets:
    print("Euclid is a pet!")
```

Sets

Create a set or convert a list to a set

```python
my_pet_list = ['euclid', 'leta']

# you can convert lists to sets using the set keyword
my_pet_set = set(my_pet_list)

# sets are like lists but you can't index into them or slice them
# they are used for fast membership testing

# you can create a new set by:
my_pet_set = set(['euclid', 'leta'])
```

Add an item to a set

```python
my_new_set = set()

# instead of append, like a list, you use 'add'
my_new_set.add("Potatoes")
```

Using sets to enforce uniqueness

```python
my_grocery_list = ['potatoes', 'cucumbers', 'potatoes']

# now if you want to make sure items only appear once, you can convert it to a set
# it will automatically do this for you, because items are only allowed to be in sets␣
↪one time

my_grocery_set = set(my_grocery_list)
```

Conditionals

If, elif, and else

Use an if to test for something

```python
power_level = 1000
min_power_level = 500
max_power_level = 1000

# one thing is larger than another
if power_level > minimum_power_level:
    print("We have enough power!")

if power_level == max_power_level:
    print("You have max power!")
```

Create conditional logic

```python
selected_option = 2

if selected_option == 1:
    print("Doing option 1")
elif selected_option == 2:
    print("Doing option 2")
elif selected_option == 3:
    print("doing option 3")
else:
    print("Doing the default option!")
```

Nest one if inside another if

```python
name = "euclid"
animal = "bunny"

if animal == "bunny":
    if name == "euclid":
        print("Euclid is my bunny")
    elif name == "leta":
        print("Leta is my bunny")
    else:
        print("this is not my bunny..")
else:
    print("Not my animal!")
```

Loops

For Loops

Write a for loop

```python
for i in range(10):
    print("do stuff here")
```

Use the for loop's loop variable

```python
for i in range(10):
    new_number = i * 100
    print("The loop variable is i.  It equals {}".format(i))
    print("I used it to make a new number. That number is {}".format(new_number))
```

Use range inside a for loop

```python
start = 3
stop = 10
step = 2

for i in range(stop):
    print(i)

for i in range(start, stop):
    print(i)

for i in range(start, stop, step):
    print(i)
```

Use a list inside a for loop

```python
my_pets = ['euclid', 'leta']

for pet in my_pets:
    print("One of my pets: {}".format(pet))
```

Nest one for loop inside another for loop

```python
for i in range(4):
    for j in range(4):
        result = i * j
        print("{} times {} is {}".format(i, j, result))
```

While Loops

Use a comparison

```python
response = ""

while response != "exit":
    print("Inside the loop!")
    response = input("Please provide input: ")
```

Use a boolean variable

```
done = False

while not done:
    print("Inside the loop!")
    response = input("Please provide input: ")
    if response == "exit":
        done = True
```

Loop forever

```
while True:
    print("Don't do this!  It is a bad idea.")
```

Special Loop Commands

Skip the rest of the current cycle in the loop

```
for i in range(100):
    if i < 90:
        continue
    else:
        print("At number {}".format(i))
```

Break out of the loop entirely

```
while True:
    response = input("Give me input: ")
    if response == "exit":
        break
```

Functions

The syntax for a function

The first line of a function, called the function header, requires the following:

1. the `def` keyword,
2. the name
3. parenthesis
4. a colon

For example:

```
def some_name():
```

- Notice that there is a space between the `def` and the function name, `some_name`.
- There is also NO space between `some_name` and the parenthesis.
- In the most basic version, there is nothing inside the parenthesis.
- Immediately following the parenthesis (with NO space!), there is a colon.

You are free to name functions whatever you want. Everything else must remain the same!

Go to the next section to see how we put code inside a function.

No arguments and returns nothing

When there is nothing inside the parenthesis in a function header, we say that the function takes no arguments. When it doesn't send back a value, we say it returns nothing.

For example:

```
def say_hello():
    print("hello!")
```

Now, if you want to call the function, go to the next part!

Calling a function you wrote

When you write a function, you give it a name. For example, in the section just before this one, the function name is `say_hello`.

If you have a python file where you have defined that function, you can then call it. Calling a function looks just like the following:

```
1   ### this is the definition! It does not call the function
2   def say_hello():
3       ## this comment is inside the function, because of the indentation!
4       print("hello!")
5       ## this comment is also inside the function
6
7   ### this comment is outside the function!  The indentation is gone
8
9   ### the function has to be called outside of the function!!
10  ### We use its name plus parenthesis
11  say_hello()
```

Notice a couple of things:

1. The name of the function is `say_hello`.
2. **At the end of the code above, on line 11, we use the function.**
    - This is also known as **executing** or **calling** a function.
3. **Python knew to call the function because of the parenthesis in line 11.**
    - It is part of an agreement with Python and programmers that programmers will use

    parenthesis in this way, immediately after a function name, to tell it that it wants to call a function.

Takes one argument

Now, let's look at how a function can have a single argument. This is why there are parenthesis in the function header. This lets specify what arguments a function will have.

```python
def say_something(the_thing):
    print("2. I will say something now!")
    print(the_thing)
    print("4. I just said something!")

print("1. I am going to call the say_something function!")
say_something("3. This is cool!")
print("5. I just called the function!")
```

In this example, there are a lot of print statements! Run the code and see the order in which they print out. I have numbered the print statement so you can see the order.

**The important thing to know: **

- Once Python "enters" into the function to start running the code, it is

in a local context - This means that the variable named `the_thing` exists only inside the function - It is a temporary variable Python makes to hold the value you pass in when you call the function.

Returns a value

Now let's look at how you can **return** items from a function!

```python
def double(x):
    return 2*x
```

Takes two arguments

```python
def exp_func(x, y):
    result = x ** y
    return result

final_number = exp_func(10, 3)
```

Takes keyword arguments

```python
def say_many_times(message, n=10):
    print("Inside the say_many_times function!")
    for i in range(n):
        print(message)


say_many_times("Hi!", 2)
say_many_times("Yay!")
```

Class Basics

First class

Defining class is a recipe. Take a look at the syntax:

```python
class Dog:
    name = 'default name'
    age = 0
```

The important part to notice is the `class Dog:`. This is what indicates the beginning of the code block. After the class is defined, two variables are declared. These variables are inside the class. Think of them like files in a folder.

```python
class Dog:
    name = 'default name'
    age = 0

fido = Dog()
```

This code **instantiates** the class. This means you are using the recipe to create a new object.

To repeat the vocabulary:

- **instantiate**: use the recipe to create an object
- **object**: a specific instance of a class. think of this like a cookie from a cookie recipe.

Using the first class

```python
class Dog:
    name = 'default name'
    age = 0

fido = Dog()

print("1. Fido's name: ", fido.name)
fido.name = "Fido"

print("2. Fido's name: ", fido.name)

george = Dog()
print("3. George's name: ", george.name)
print("3. Fido's name: ", fido.name)

george.name = "George"
print("4. George's name: ", george.name)
print("4. Fido's name: ", fido.name)
```

Run this example.

You will see that changing the internal properties of fido and george stay inside fido and george! This is another example of scope. Just like inside functions are local scope, inside objects are local scope!

Getting access to internal variables

```python
class Dog:
    name = 'default name'
    age = 0

def speak(some_dog):
    print("My name is {}. Bow wow!".format(some_dog.name))

fido = Dog()
speak(fido)
```

We can write functions which use the `fido` object as its argument and access the internal variables!

Getting access to internal variables

You can see from the last example that you access the internal variables using the dot notation. But, what if you wanted to write a function *inside* the object? How can you access the variables?

Let's try this:

```python
class Dog:
    name = 'default name'
    age = 0

    def speak():
        print("My name is {}. Bow wow!".format(name))

fido = Dog()
fido.speak()
```

Do you think this will work? Nope! Scope doesn't let us do that!

There is a second reason why the code above won't work and that reason is also what solves things!

```python
class Dog:
    name = 'default name'
    age = 0

    def speak(self):
        print("My name is {}. Bow wow!".format(self.name))

fido = Dog()
fido.speak()
```

When you use the function that is inside an object, python adds a variable without you having to do anything! That variable is called the `self` variable. This is just like having the function outside of the `class`, except that Python puts the `self` variable there automatically, so we don't have to.

`def __init__(self)`

The `__init__` function is one of Python's special functions - this is indicated by the double underscore (__) on either side of the function name. `init` is a keyword (like `print` or `if``) and Python already knows what it's used for.

When you write your own class, sometimes it's helpful to have a kind of setup function that runs whenever you make a new copy of the class. For example, if you write the `Door` class we've been using as an example, you might want the `Door` to print out "Hello!" the first time someone makes it. And, every new `Door` that gets made will also say "Hello!"

This is what the `__init__` function is for: it's a special function that runs once every time an object of that type (in our example, `Door`) is made.

So, for example:

```python
class Door:
    def __init__(self):
        print("Hello!")

first_door = Door()
second_door = Door()
```

The code above will print out "Hello!" twice - once for `first_door`, and again for `second_door`.

That's an example of an `__init__` function that doesn't take any arguments. Usually, this isn't the case - because `__init__` is a setup function, you want the user to provide certain information about the object when they make it.

Here's an example:

```python
class Door:
    def __init__(self, in_name, in_height):
        self.name = in_name
        self.height = in_height
        print("Hello! My name is " + self.name)

first_door = Door("Gerald", 10)
second_door = Door("Geraldina", 12)
```

In this code, when a `Door` object is created, it takes two arguments: the name, and the height. These arguments are then used for setting up the Door object (i.e., they set up the properties `self.name` and `self.height`)

Advanced Classes

Design patterns and examples for classes! Use these to help you solve problems.

Defining a class

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Instantiating an object

```python
# create the object!
fido = Dog("Fido", 7)
```

Writing a method

A method is the name of a function when it is part of a class.

You always have to include `self` as a part of the method arguments.

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Bow wow!")


fido = Dog("Fido", 7)
fido.bark()
```

Using the self variable

You can access object variables through the `self` variable. Think of it like a storage system!

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("{}: Bow Wow!".format(self.name))


fido = Dog("Fido", 7)
fido.bark()

odie = Dog("Odie", 20)
odie.bark()
```

Using the property decorator

You can have complex properties that compute like methods but act like properties. Properties cannot accept arguments.

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

fido = Dog("Fido", 7)
```

```python
fido.bark()
print("Fido is {} in human years".format(fido.human_age))
```

Inheriting properties and methods

You can inherit properties and methods from the ancestors! For example, the initial function below is inherited.

```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Dog(Animal):
    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

class Cat(Animal):
    def meow(self):
        print("{}: Meow!".format(self.name))

fido = Dog("Fido", 7)
fido.bark()
print("Fido is {} in human years".format(fido.human_age))
```

You can also override certain things and call the methods of the ancestor!

```python
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age):
        super(Dog, self).__init__(name, age, 4, "dog")

    def make_noise(self):
        self.bark()

    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

class Cat(Animal):
    def __init__(self, name, age):
        super(Dog, self).__init__(name, age, 4, "cat")

    def make_noise(self):
        self.meow()

    def meow(self):
        print("{}: Meow!".format(self.name))


fido = Dog("Fido", 7)
fido.make_noise()
print("Fido is {} in human years".format(fido.human_age))

garfield = Cat("Garfield", 5, 4, "cat")
garfield.make_noise()
```

Using the classmethod decorator

There is a nice Python syntax which lets you define custom creations for your objects.

For example, if you wanted certain types of dogs, you could do this:

```python
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age, breed):
        super(Dog, self).__init__(name, age, 4, "dog")
        self.breed = breed


fido = Dog("Fido", 5, "Labrador")
```

But you could also do this:

```python
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age, breed):
        super(Dog, self).__init__(name, age, 4, "dog")
        self.breed = breed

    @classmethod
    def labrador(cls, name, age):
        return cls(name, age, "Labrador")

fido = Dog.labrador("Fido", 5)
```

Important parts:

1. **Instead `self`, it has `cls` as its first argument.**
    - This is a variable which points to the class being called.
2. **`@classmethod` is right above the definition of the class.**
    - It absolutely has to be exactly like this
    - No spaces in between, just sitting on top of the class definition
    - It's called a decorator.
3. **It returns `cls(name, age, "Labrador")`.**
    - This is exactly the same as `Dog("Fido", 5, "Labrador")` in this instance
    - Overall, it is letting you shortcut having to put in the labrador string.

This is a simple example, but it is useful for more complex classes

Built-in Functions

Built-in Functions

```python
print("This prints to the console/terminal!")

# notice the space at the end!
# it helps so that what you type isn't right next to the ?
name = input("What is your name? ")

# use input to get an integer
age = input("How old are you?")
# but it's still a string!
# convert it
age = int(age)

# test the length of a list or string
name_length = len(name)

# get the absolute value of a number
positive_number = abs(5 - 100)

# get the max and min of two or more numbers
num1 = 10**3
num2 = 2**5
num3 = 100003
biggest_one = max(num1, num2, num3)
smallest_one = min(num1, num2, num3)
# can do any number of variables here
#   max(num1, num2) works
#   and max(num1, num2, num3,  num4)

## max/min with a list
ages = [12, 15, 13, 10]
min_age = min(age)
max_age = max(age)

# sum over the items in a list
# more list stuff is below
ages = [12, 15, 13, 10]
sum_of_ages = sum(ages)
number_of_ages = len(ages)
average_age = sum_of_ages / number_of_ages
```

Built-in Modules

Time module

Using time.time() to count how long something takes

```python
import time

start = time.time()

for i in range(10000):
    continue

new_time = time.time()
total_time = new_time - start
print(total_time)
```

Using time.sleep(n) to wait for n seconds

```python
import time

start = time.time()

time.sleep(10)

end = time.time()

print(start - end)
```

Random Module

Generate a random number between 0 and 1

```python
import random

```

```python
num = random.random()
print("the random number is {}".format(num))
```

Generate a random number between two integers

```python
import random

num = random.randint(5, 100)
print("the random integer between 5 and 100 is {}".format(num))
```

Select a random item from a list

```python
import random

my_pets = ['euclid', 'leta']
fav_pet = random.choice(my_pets)
print("My randomly chosen favorite pet is {}".format(fav_pet))
```

Read and Write Files

It is common to read and write files to store and process data and information. This could the saving of a game or processing a file.

There are a couple new syntax items we haven't talked about yet.

1. `with` is Python syntax that temporarily creates variables and handles the creation and closing. We use it with files so Python makes sure the file closes when we are done and doesn't corrupt the file. `with` will create a new code block, so it's important to note that as soon as the code block ends (a line of code starts that isn't indented the 4 spaces needed to be in the `with` code block), then Python closes the file.

2. `open` is a Python command to open a file. With only one argument, that argument is assumed to be the filepath. There are optional arguments. The important one is a second argument which will tell Python whether you are reading or writing the file. When you don't put in the second argument, Python assumes it is `'r'` for reading.

3. `as` is Python syntax only used with the `with` command. You should keep note of the syntax as a whole and follow that pattern.

Reading the file

```python
filepath = "file_location.txt"
with open(filepath) as file_obj:
    data = file_obj.read()
```

in this example, we knew where the location of the file is. This is vital and there are two different ways to think about this.

1. **The relative path is based on the file that's running the code (or where** the iPython console thinks it is currently at)
2. You have to know the **absolute path**, which is the fully path to the file.

I recommend trying this out with a file that's in the same folder as the python file. This is the **relative path** style. Then the filepath is just the filename.

The hidden second argument to `open`

As mentioned above, `open` can have a second argument. In this example, the second argument is not there and Python has a default value for it. To explicitly put it there so we can see the value, it would look like:

```python
filepath = "file_location.txt"
with open(filepath, 'r') as file_obj:
    data = file_obj.read()
```

The `'r'` stands for **read**

Variations on read

There are a couple of different ways you can read in the file. The example above showed the function `file_obj.read()`. This gets the entire file in as a string. Another way is to use `file_obj.read_lines()`. If you're doing line by line processing, I recommend the second. If not, I recommend the first.

Things to be careful of

Strings can be tricky. They can include characters that you don't want. For example, with `file_obj.read()`, all of the new lines (the characters that give new lines when printed) are still in there (as `\n`). You can fix this by removing them from the string using the replace function:

```python
x = "this has \n\tweird characters"

x = x.replace("\n", "")
x = x.replace("\t, "")
print(x)
assert x == "this has weird characters"
```

Writing to files

Writing to files is very similar. The important things to notice:

1. There is now an `'w'` in the syntax: `with open(filepath, 'w') as file_obj`. This means Python is now writing to the file. Be careful though! **This will always overwrite any file that is already there!**

2. You have to write the new lines yourself (otherwise everything will be on one line!)

```python
filepath = "file_location.txt"
with open(filepath, 'w') as file_obj:

    file_obj.write("This is in the file")
    file_obj.write("So is this! But this one has the new line\n")
    file_obj.write("Here is some more stuff")
```

Reading and Writing JSON Files

JSON files are really useful. They let you store not just plain text, but dictionaries! You pronounce them as “Jay-Sawn” (Jason is more like “Jay-Sun”, which isn’t quite right here)

Writing a JSON file

Writing a JSON file is called “dumping”.

```python
import json

bunny_dict = {"name": "Euclid", "age": 2}

with open("bunny.json", "w") as fp:
    json.dump(bunny_dict, fp)
```

And that’s it! One nice part about JSON is it that you can dump a JSON dictionary to a string instead of a file!

```python
import json

bunny_dict = {"name": "Euclid", "age": 2}

new_str = json.dumps(bunny_dict)

print(new_str)
```

Notice that it is `json.dumps` now. The **s** in `dumps` means “dump string”.

Reading a JSON file

Loading a JSON file is called “loading”

```python
import json

with open("bunny.json", "r") as fp:
    bunny_dict = json.load(fp)
```

Pretty easy! The nice part about using JSON is that you can use dictionaries to store information. This means you can store and load information which requires more structure than a plain string, like character information or save game state.

CHAPTER 2

Tutorials

Planning with Flow Charts and Structure

Flow charts and structure

You should decide on the structure and topic of your project pretty early.

For example:

- if you are doing a story, what is the context and background?
- if you are doing a game, what is the goal? what are the parts?
- similar to a game, if you are doing a library (like encryption), what is the goal?

A flow chart starts with the initial point. For an interactive story or game, this is the beginning of the game. For a library, it will be the beginning of the algorithm. The logic of the code will flow from this initial point. You can use either paper to draw this or you can use an online website (for example, draw.io is an ok one).

Any specific point in the story is sometimes called the "state". The state is a specific setting of variables. And since this is a story you are programming, the set of variables are the variables you will be using to keep track of the story.

Remember that you are drawing the information flow. Here is a real life example.. Though, they don't use the dimaonds I recommend below, but instead use colors. You can do whichever you want.

This page has really good, simple examples of flow charts

There are several kinds of shapes in the flow charts:

Ovals are start/end points

Ovals are either the starting or ending points. They are where the story starts and stops.

Rectangles/Boxes are processing points

Boxes represent the processing of information. Into the box flows some information and out flows other information. You draw this as a line flowing into a box and a line flowing out.

Diamonds are decision points

If there is a condition in your code, you draw that as a diamond. Since the information flow is being drawn as lines, a line connect to the diamond. It is good to draw an arrow on the line to show which direction it is flowing.

Each choice is a different path from the diamond. I usually try to keep my diamonds as True/False and have a path come off one side of the diamond and off the point opposite from where the information flow entered the diamond.

Other shapes

There are a lot of shapes people use and slightly different ways to use them. You have complete freedom to represent other types of information flow with other shapes. Just make sure you use them the same way everywhere. I personally only work with these three shapes.

Interactive Stories and Games

Understanding State

The word state refers to a specific setting of variables. In practice, there are several different ways you could accomplish this. It is important to think about it in the following way: in the flow chart, you are defining how information flows from state to state, but in the code, you should be trying to design code that works similarly for every state.

The simplest state, for example, is to have just a single number which represents which state you are in. Then, when you need to have code that is conditional on the state, you could do:

```python
state = 1 # for example

if state == 0:
    print("In state 0")
if state == 1:
    print("In state 1")
if state == 2:
    print("In state 2")
```

The Main Loop

When doing things like stories and games, you need to have a loop which manages the repeated behaviors. In other words, you make code that handles all states, and you use a while loop which keeps looping over the code.

Here is an example. I used `state` as an integer to keep track of what point the user is in the story.

```python
game_over = False
state = 0

while not game_over:

    if state == -1:
        game_over = True
    elif state == 0:
        print("Welcome to the dungeon..")
        print("You are in a dark room. Do you go left or right?")
        choice = input("> ")
        if choice == "left":
            state = 1
        elif choice == "right":
            state = -1
            print("You ran into a monster and died!")
        else:
            print("I don't understand that command!")
    elif state == 1:
        print("more code here and for other states!")
```

While the above code is good and you could write the entire game like that, you have two choices for how to make it more compact, easier to write, and easier to use.

The first is to use functions. For this method, you would have a test to see which state you are in and then call different functions based on that state.

The second is to use classes. For this method, you have objects can accomplish the actions of the different states, but they have a shared function. That way, from the perspective of the main loop, the same function is being called every time.

Using Functions

The game loop above could get really messy and long. One way to manage the top-level view without having too crazy of code is to break parts up into states. For example, you could put the above first-state code into a function

```
def state_0():
    print("Welcome to the dungeon..")
    print("You are in a dark room. Do you go left or right?")
    choice = input("> ")
    if choice == "left":
        state = 1
    elif choice == "right":
        state = -1
        print("You ran into a monster and died!")
    else:
        print("I don't understand that command!")

    return state

### the main game loop
state = 0
game_over = False
while not game_over:
    if state == -1:
        game_over = True
    elif state == 0:
        state = state_0()
    elif state == 1:
        print("more stuff")
```

Notice how cleaner this code is! Try to write state functions so that your code stays clean. It is good practice to break code into chunks like this.

Notice that the function returns the `state` integer. The reason you'd want to return this is because then you're letting each state decide whether the game ended. You could do it another way, if you wanted. For example, you could have player information like health. And then, you pass that information into the function and pass it back out. You could then check to see if player health was 0 and that would end the game.

More complex states

You could also be writing more complex states. For example, you could be using a dictionary or a list with information in it.

```
euclid = {'location': 'kitchen', 'health': 10, 'name': 'Euclid'}
```

You could create classes for things like this:

```
class Bunny:
    def __init__(self, name, location, health):
        self.name = name
        self.location = location
        self.health = heatlh

euclid = Bunny("Euclid", "kitchen", 10000)
```

An example

```
class Human:
    def __init__(self, name, health, location):
        self.name = name
        self.health = health
        self.location = location
        self.thing_type = 'human'


class Animal:
    def __init__(self, name, health, location, animal_type):
        self.name = name
        self.location = location
        self.health = health
        self.thing_type = "animal"
        self.animal_type = animal_type

def get_response(options):
    while True:
        print("Options: ")
        for option in options:
            print("\t{}".format(option))
        user_choice = input("What do you do? ")

        if user_choice not in options:
            print("'{}' is not a valid option, try again".format(user_choice))
        else:
            return user_choice

def kitchen(player, world):
    print("You are in the kitchen.. it is very spooky and very scary")

    animals_in_kitchen = list(world.animals['kitchen'].items())

    if len(animals_in_kitchen) == 0:
        print("Nothing in the kitchen.. how boring.")
        print("I wish you could go to other rooms now...")

    for animal_name, animal in animals_in_kitchen:
        print("Suddenly, a {} comes out of no where!".format(animal.animal_type))

        options = ['pet', 'run away', 'hide']

        choice = get_response(options)

        if choice == 'pet':
            print("You tried to pet him, but he bit your finger!")
            player.health = 0
        elif choice == "run away":
            print("You successfully ran away into the garage")
            player.location = "garage"
        elif choice == "hide":
            print("You hide under the table!")
            world.move_animal(animal, from_location="kitchen", to_location="hidden")


def garage(player, world):
    print("You are in the garage.. and you're locked in. whoops!")
    print("You are stuck here forever")
    player.health = 0

class World:
    def __init__(self):
        self.locations = {"hidden": None}
        self.animals = {"hidden": {}}
        self.player = None

    def put_in_world(self, new_thing, thing_type):
        location = new_thing.location
        if location not in self.locations.keys():
            raise Exception("Location '{}' hasn't been added yet!!".format(location))
        if thing_type == 'animal':
            ## note: you can't have the same names, it will overwrite!
            self.animals[location][new_thing.name] = new_thing
        elif thing_type == 'player':
            self.player = new_thing
        else:
            raise Exception("Not supported yet: {}".format(thing_type))

    def add_location(self, location, location_function):
        self.locations[location] = location_function
        self.animals[location] = dict()

    def move_animal(self, animal, from_location, to_location):
        del self.animals[from_location][animal.name]
        self.animals[to_location][animal.name] = animal

euclid = Animal(name="Euclid", health=10, location="kitchen", animal_type='bunny')
player = Human(name="Mr. McMahan", health=100, location="kitchen")

```

```python
### this is putting the function inside the dictionary!
world = World()

world.add_location('kitchen', kitchen)
world.add_location('garage', garage)
world.put_in_world(player, 'player')
world.put_in_world(euclid, 'animal')


game_active = True

max_num_turns = 100

while game_active:
    location = player.location
    location_function = world.locations[location]

    location_function(player, world)

    if player.health == 0:
        print("Player has passed out! The game was too much. game over")
        game_active = False
    else:
        options = ['leave', 'keep going']

        choice = get_response(options)

        if choice == 'leave':
            print("Goodbye!")
            game_active = False
```

Ciphers

Substitution Cipher

There are many different kinds of ciphers. A simple one is a substitution cipher. This is where you create a dictionary that maps one letter to another. Then, you can loop over the message and make a new one!

Since we are programmers, we don't always want to make things from scratch. Let's look at some shortcuts too.

```python
import string

print("the string package has a bunch of useful things for strings in Python")
print("We are going to use it to get all of our letters")

letters = string.ascii_lowercase

print(letters)

cipher = dict() ## an empty dictionary

n_letters = len(letters)

for i in range(n_letters):
    from_letter = letters[i]

    ### now, let's get the opposite letter
    ## n_letters is 26, and i starts at 0
    ## but we want the first to_letter_index to be 25. so let's subtract 1.
    ## what is the last number i will be?  what will this index be then?
    to_letter_index = n_letters - i - 1
    to_letter = letters[to_letter_index] ### this will start at n_letters and work backwards!

    cipher[from_letter] = to_letter

print("We have now made our cipher, let's use it to encrypt a message!")

message = "this is a test Message"

print("Because we only made our cipher with lowercase letters, let's make sure the message is lowercase too")

message = message.lower()

print("Now we are going to use the accumulator pattern.")
print("This means we start with an empty string and add onto it, one letter at a time")

encrypted_message = ""
for letter in message:

    ### make sure to test to see if the letter is even in the cipher. if it's not
    ### it is a space or a number. let's leave them along and just add them into
    ### the resulting string!
    if letter in cipher.keys():
        new_letter = cipher[letter]
    else:
        new_letter = letter ## just set it equal it self

    encrypted_message += new_letter ## remember the += is a shortcut!

print("We have encrypted the message {} and it has become {}".format(message, encrypted_message))
```

Decrypting

Decrypting is the opposite of encrypting: you just map backwards.

One way to do this is to just use the dictionary we made and reverse it!

```python
import string
letters = string.ascii_lowercase
n_letters = len(letters)

cipher = dict() ## an empty dictionary
decrypt = dict() ## an empty dictionary

for i in range(n_letters):
    from_letter = letters[i]
    ### now, let's get the opposite letter
    ## n_letters is 26, and i starts at 0
    ## but we want the first to_letter_index to be 25. so let's subtract 1.
    ## what is the last number i will be?  what will this index be then?
    to_letter_index = n_letters - i - 1
    to_letter = letters[to_letter_index] ### this will start at n_letters and work␣
↪backwards!

    cipher[from_letter] = to_letter
    decrypt[to_letter] = from_letter

## you could have also done this:
decrypt = dict()
for from_letter, to_letter in cipher.items():
    decrypt[to_letter] = from_letter

### and now you use it as we did with the cipher above
```

Caesar Cipher

A really awesome fact is that the Caesar Cipher is named after Ceasar himself.

From the Wikipedia page:

> The Caesar cipher is named after Julius Caesar, who, according to Suetonius, > used it with a shift of three to protect messages of military significance. > While Caesar's was the first recorded use of this scheme, other substitution ciphers are known to have been used earlier.[4][5]

> "If he had anything confidential to say, he wrote it in cipher, > that is, by so changing the order of the letters of the alphabet, > that not a word could be made out. If anyone wishes to decipher these, > and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others." > - Suetonius, Life of Julius Caesar 56

> His nephew, Augustus, also used the cipher, but with a right shift of one, and it did not wrap around to the beginning of the alphabet: > "Whenever he wrote in cipher, he wrote B for A, C for B, and the rest of the letters on the same principle, using AA for Z." > - Suetonius, Life of Augustus 88

> Evidence exists that Julius Caesar also used more complicated systems,[6] and one writer, Aulus Gellius, refers to a (now lost) treatise on his ciphers: > "There is even a rather ingeniously written treatise by the grammarian Probus concerning the secret meaning of letters in the composition of Caesar's epistles." > - Aulus Gellius, Attic Nights 17.9.1–5

A Caesar Cipher is just a substitution cipher which has a specific shift of letters!

So, let's assume we will use the exact same code above, but let's change how we make the cipher dictionary. The following code should only replace the `for loop` above that makes the cipher dictionary.

```
import string
letters = string.ascii_lowercase
n_letters = len(letters)

caesar_cipher = dict() ## an empty dictionary
shift_number = 3 ### Let's shift like Caesar!

for i in range(n_letters):
    from_letter = letters[i]

    ### now, let's get the encrypting letter
    ### but this time, let's use the cipher!
    to_letter_index  = i + 3

    ### but what if other_index is larger than 26 now?  We need to make sure
    ### it wraps back around to the length of the alphabet:
    to_letter_index = to_letter_index % 26

    ### now store it!
    to_letter = letters[to_letter_index] ### this will start at n_letters and work␣
↪backwards!

    cipher[from_letter] = to_letter

print("We have now made the Caesar Cipher, let's use it to encrypt a message!")

message = "this is a test Message"

print("Because we only made our cipher with lowercase letters, let's make sure the␣
↪message is lowercase too")

message = message.lower()

print("Now we are going to use the accumulator pattern.")
print("This means we start with an empty string and add onto it, one letter at a time
↪")

encrypted_message = ""
for letter in message:

    ### make sure to test to see if the letter is even in the cipher. if it's not
    ### it is a space or a number. let's leave them along and just add them into
    ### the resulting string!
    if letter in cipher.keys():
        new_letter = cipher[letter]
    else:
        new_letter = letter ## just set it equal it self

    encrypted_message += new_letter ## remember the += is a shortcut!

print("We have encrypted the message {} and it has become {}".format(message,␣
↪encrypted_message))
```

ROT-13

When the shift is 13, it is called **ROT-13** for “rotation 13”. Although not that secure (it can be decoded easily), it does provide a nice example because both the encryption and decryption dictionaries would be the same, at least for English because we have 26 letters.

To read more about it, check out the wikipedia entry. You should try the ROT-13 in your Caesar Cipher.