cs50P

cs50p.pdf

▼ 0. Functions

hello.py

we are going to use vscode

```
code hello.py
```

Now lets write our first function

```
print('hello, world')
```

to run our code in terminal

```
python hello.py
```

`print` is just a function, it takes **arguments**

remember that printing is just a **side effect**

when stuff go wrong, its called a **bug.**

return values and variables

```
 # Ask user for their name
name = input("what's your name? ")
```

```
#say hello to user
print("hello, " + name)
```

we need to save the return value to a variable called name

variables should be named well

here are the wrong ways

```
 # Ask user for their name
name = input("what's your name? ")
#say hello to user
print("hello")
print(name)
```

notice that `#` is a comment to help the user understand, its not read by the computer

to have a multiline comment

```
'''
this is blabla
'''
```

multiple functions and arguments

in py, we can do the same thing in multiple ways

concatenation

```
name = input("what's your name? ")
#say hello to user
print("hello, " + name)
```

or as another argument

```
name = input("what's your name? ")
#say hello to user
```

```
print("hello,", name)
```

or

```
name = input("what's your name? ")
#say hello to user
print("hello,",end=" ")
print(name)
```

notice that end is a **parameter**

*objects means it can take unlimited amount of object

we can also adjust sep parameter

`\n` means new line

to print “ in py

```
print("hello, \"friend\"") #hello "friend"
```

or

```
print('hello, "friend"')
```

f string

this is the new fancy way to solve our problem

```
print(f"hello, {name}")
```

string methods

till now, we were dealing with text aka string

we can clean strings using methods

```
name = input("what's your name? ").strip().title()
#say hello to user
print(f"hello, {name}")
```

we can also say

```
name = input("what's your name? ")
name = name.strip()
name = name.title()
print("hello,",end=" ")
print(name)
```

first way is better, but don't make it too complex

as long as you can argue why you are good, its ok to do what you want

we can get two outputs from the same method by the way

```
name = input("what's your name? ").strip().title()
first, last = name.split( )
#say hello to user
print(f"hello, {last}")
```

int

now lets deal with numbers, we can add subtract, …

we can do it in the **interactive mode in terminal**

```
python
1+1
```

calculator.py

```
x = int(input("what is x? "))
y = int(input("what is y? "))
```

```
print(x+y)
```

note that we must use the function `int` cuz input returns string not integers

we can also say

```
x = input("what is x? ")
y = input("what is y? ")
z = int(x) + int(y)
print(z)
```

but we are using z for one line, not useful so first version is better

we can go super crazy

```
print(int(input("what is x? ")), int(input("what is y? ")))
```

which is super complex, long, horrible

float

aka decimals

```
x = float(input("what is x? "))
y = float(input("what is y? "))

print(x + y)
```

we can round too

`[]` in documentation means its optional

```
x = float(input("what is x? "))
y = float(input("what is y? "))

z = round(x + y)
```

```
print(z)
```

floats can handle with int

in USA, we write 1000 as 1,000. to do so

```
x = float(input("what is x? "))
y = float(input("what is y? "))
z = round(x+y)
print(f"{z:,}") #999+1 = 1,000
```

it looks creepy, but it works

```
 x = float(input("what is x? "))
y = float(input("what is y? "))
z = x/y
print(round(z,2))
print(f"{z:.2f}")
```

both do the same thing

def

lets define our own function hello

we added a default argument

```
def hello(to="world"):
    print("hello,", to)


hello()
```

we indent 4 spaces

we can overwrite the default

```
name = input("what is your name? ")
hello(name)
```

its better to have our function down inside, like in line 50, but this will throw an error

Instead, we split our code into functions. main code in function main

```
def main():
    name = input("what is your name? ")
    hello(name)

def hello(to="world"):
    print("hello,", to)
main()
```

scope

remember, if a variable is in main only, its not in hello

```
def main():
    name = input("what is your name? ")
    hello()

def hello():
    print("hello,", name)
main() #error
```

This is why we have to hand the value from main to hello

```
def main():
    name = input("what is your name? ")
    hello(name)

def hello(to="world"):
```

```
    print("hello,", to)
main()
```

return

remember that our code is doing a side effect aka printing,

lets fix it

```
def main():
x = int(input("whats x?"))
    print("x squared is", square(x))

def square(n):
    return pow(n,2)
main()
```

🚀 in the main, we act as if our function does exist before we even write it

▼ 1. Conditionals

do this line of code or this?

we can use `>, >= < <= == !=`

if

```
x = int(input("what is x? "))
y = int(input("what is y? "))

if x < y:
    print("x is less than y")
if x > y:
    print("x is greater than y")
```

```
if x == y:
    print("x is equal to y")
```

x > y returns a B**oolean value** aka true or false

This code actually tests all three if statements. which is not efficient

why ask three questions!

elif

```
x = int(input("what is x? "))
y = int(input("what is y? "))

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
elif x == y:
        print("x is equal to y")
```

elif only works if the previous if or elif did not run

do we really to write the last question?

> nope, its the only option remaining, so its redundant

else

```
x = int(input("what is x? "))
y = int(input("what is y? "))

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
```

```
else:
    print("x is equal to y")
```

or

we can ask two questions at the same time, aka two conditions

```
x = int(input("what is x? "))
y = int(input("what is y? "))

if x < y or x > y:
    print("x is not equal to y")
else:
    print("x is equal to y")
```

but why ask two questions if we just want to ask if they are equal or not

```
x = int(input("what is x? "))
y = int(input("what is y? "))

if x != y:
    print("x is not equal to y")
else:
    print("x is equal to y")
```

and

again, two conditions but both of them must be true

```
score = int(input("score: "))

if score >= 90 and score <= 100:
    print("Grade: A")
elif score >= 80 and score < 90:
    print("Grade B")
```

```
elif score >= 70 and score < 80:
    print("Grade C")
elif score >= 60 and score < 70:
    print("Grade D")
else:
print("Grade: F")
```

OMG, this was a lot. We can write it in a better way

```
score = int(input("score: "))

if 90 <= score <= 100:
    print("Grade: A")
elif 80 <= score <= 90:
    print("Grade B")
elif 70 <= score <= 80:
    print("Grade C")
elif 60 <= score <= 70:
    print("Grade D")
else:
    print("Grade: F")
```

py allows us to write it like this

But we can write our program in a smarter way

```
score = int(input("score: "))

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade B")
elif score >= 70:
    print("Grade C")
elif score >= 60:
    print("Grade D")
```

```
else:
    print("Grade: F")
```

Now its shorter and we ask one question only

modulo operator %

which gets the remainder. we can use it to get cool programs

```
x = int(input("what is x? "))

if x % 2 ==0:
    print("Even")
else:
    print("Odd")
```

if no remainder, then its an even number

Remember that its better to use our functions

```
def main():
    x = int(input("what is x? "))
    if is_even(x):
        print("Even")
    else:
        print("Odd")


def is_even(n):
    if n % 2 == 0:
        return True
    else:
    return False

main()
```

is_even returns True or False, which is the answer to our condition

so we abstracted away the condition part

pythonic expressions

which are ways of writing the code that are ultra awesome but not in other programming languages

```
def main():
    x = int(input("what is x? "))
    if is_even(x):
        print("Even")
    else:
        print("Odd")


def is_even(n):
    return True if n % 2 == 0 else False

main()
```

it reads like English. But we can even make it better

```
def main():
    x = int(input("what is x? "))
    if is_even(x):
        print("Even")
    else:
        print("Odd")


def is_even(n):
    return  % 2 == 0

main()
```

match

this was added in 3.10. similar to switch in other languages

```
name = input("what is yout name? ")

if name == "Harry":
    print("Gryffindor")
elif name == "Hermione":
    print("Gryffindor")
elif name == "Ron":
    print("Gryffindor")
elif name == "Draco":
    print("Slytherin")
else:
print("who?")
```

See the redundancy? better to ask multiple condition

```
 name = input("what is yout name? ")

if name == "Harry" or name == "Hermione" or name == "Ron":
    print("Gryffindor")
elif name == "Draco":
    print("Slytherin")
else:
    print("who?")
```

We can also use `match`

```
name = input("what is yout name? ")

match name:
    case "Harry":
        print("Gryffindor")
    case "Hermione":
```

```
        print("Gryffindor")
    case "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("who?")
```

we can also use or as `|`

```
name = input("what is yout name? ")

match name:
    case "Harry" | "Hermione" | "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("who?")
```

▼ 2. Loops

while

looping is the ability to do stuff again and again

```
print("meow")
print("meow")
print("meow")
```

Why repeat myself? and should I write it 50 times to print 50 times?

```
i = 3
while i != 0:


```

```
    print("meow")
    i -= 1
```

its like counting down, if I don't count down, it will run **infinitely**

to cancel `ctrl c`

Instead of counting down, we can count up

```
i = 1
while i <= 3:
    print("meow")
    i = i + 1
```

Remember: we can write `i += 1` as `i = i + 1`

Note: programmers like counting from 0

```
i = 0
while i <d 3:
    print("meow")
    i += 1
```

for

for is awesome with **lists**

lists demoted by `[]` which is a list of number or characters, etc..

```
for i in [0,1,2]:
    print("meow")
```

Think of extreme cases, if i want to print 100 times

i need a function

```
for i in range(5):
    print('meow')
```

in python, if I dont care about `i` and just using it to loop

```
for _ in range(5):
    print('meow')
```

python is cool though, I can just use this (specific to python)

```
print('meow\n' * 3, end = '')
```

validating input

If i want to force the user to enter integer number, I make him enter an infinite loop that stops only when i use **break**

```
while True:
    n = int(input("what's n? "))
    if n < 0:
        continue
    else:
        break
```

continue means enter next iteration so its redundant here

```
while True:
    n = int(input("what's n? "))
    if n > 0:
        break

for _ in range(n):
    print("meow")
```

we know, making functions is always better

```
def main():
    number = get_number()
```

```
    meow(number)


def get_number():
    while True:
        n = int(input("what's n? "))
        if n > 0:
            return n

def meow(n):
    for _ in range(n):
        print("meow")


main()
```

return breaks the loop

iterate with list

we can access items inside a list

```
students = ["Hermione","Harry","Ron"]

print(students[0])
print(students[1])
print(students[2])
```

You can see it, we can loop

```
students = ["Hermione","Harry","Ron"]

for student in students:
    print(student)
```

use `student` not `_` cuz we are using it.

len

we can iterate using another way, the number way

```
students = ["Hermione","Harry","Ron"]

for i in range(len(students)):
    print(i + 1,students[i])
```

Since we start counting from zero, but users count from 1, we print `i + 1`

later on, we will improve this function by using `enumerate`

dictionary

words and definitions or `keys` and `values`

like tracking who is in which house

```
students = ["Hermione","Harry","Ron","Draco"]
houses = ["Gryffindor","Gryffindor","Gryffindor","Slytherin"]
```

we need them to be just one thing not two

```
students = {
    "Hermione": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin"
}

print(students["Hermione"])
```

to print value, index using key

we can iterate over ***keys*** and use them to get values

```
students = {
    "Hermione": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin"
}

for student in students:
print(student, students[student],sep=", ")
```

lists of dictionaries

```
students = [
{"name":"Hermione","house": "Gryffindor","patronus":"Otter"},
{"name":"Harry","house": "Gryffindor","patronus":"Stag"},
{"name":"Ron","house": "Gryffindor","patronus":"Jaack Russel
{"name":"Draco","house": "Slytherin","patronus":None}
]

for student in students:
print(student["name"],student["house"],student["patronus"],se
```

Note: Draco has no Patronus aka None

Nested loops

Lets print mario blocks.

```
print("#")
print("#")
print("#")
```

we know there is a better way

```
for _ in range(3):
    print("#")
```

and we know I will say now: turn it into a function

```
def main():
    print_column(3)


def print_column(height):
    for _ in range(height):
        print("#")

main()
```

and we can get fancier

```
def main():
    print_column(3)


def print_column(height):
    print("#\n" * height , end="")


main()
```

Lets print horizontal rows now

```
def main():
    print_row(3)

def print_row(width):
    print("?" * width)
```

```
main()
```

But how to print a square? aka columns and rows

```
def main():
    print_square(3)

def print_square(size):
    # for each row in square
    for i in range(size):

        # for each brick in row
        for j in range(size):
            # print brick
            print("#", end ="")

        # prints new line
        print()

main()
```

we can get fancier

```
def main():
    print_square(3)

def print_square(size):
    for i in range(size):
        print("#" * size)

main()
```

and we can split square functionality into two

```
def main():
    print_square(3)

def print_square(size):
    for _ in range(size):
        print_row(size)

def print_row(width):
    print("#" * width)


main()
```

▼ 3. Exceptions

SyntaxError

```
print('hello, world)
```

notice how we did not close the string, which results in syntax errors

I need to fix it on my own

other errors, I need to program **defensively**

ValueError

```
x = int(input("what's x? "))
print(f'x is {x}')
```

now, what if the #$% user enters a string like cat

I get value error

we need to handle the error

try except

```
try:
    x = int(input("what's x? "))
    print(f'x is {x}')
except ValueError:
    print('x is not an integer')
```

we literally try the code under try, if we anticipate an error, handle it using except

Note, don’t catch all errors in a single except

Note 2: we are trying here two lines, which is bad really, try only one line

```
try:
    x = int(input("what's x? "))
except ValueError:
    print('x is not an integer')

print(f'x is {x}')
```

Name error

sadly, the previous code causes name error

why?

> x = int code is not computed aka not evaluated, then i ask for x at line 6

How to fix it?

> use else

else

```
try:
    x = int(input("what's x? "))
except ValueError:
    print('x is not an integer')
else:
    print(f'x is {x}')
```

else will run if there **are no errors**

this code ain't good though, we want to prompt the user to enter the right value

prompting

```
while True:
    try:
        x = int(input("what's x? "))
    except ValueError:
        print('x is not an integer')
    else:
        break

print(f'x is {x}')
```

This code is the best so far, I will keep asking the user until it works, then I break

we can still modify the code

```
while True:
    try:
        x = int(input("what's x? "))
        break
    except ValueError:
        print('x is not an integer')

print(f'x is {x}')
```

break here will only work if the x = int works, both versions are ok

get_int

lets create the function

```
def main():
    x = get_int()
    print(f"x is {x}")
```

```
def get_int():
    while True:
    try:
        x = int(input("what's x? "))
    except ValueError:
        print('x is not an integer')
    else:
        break
    return x


main()
```

notice how all the functionality is hidden in the function

we can make it better though

```
def main():
    x = get_int()
    print(f"x is {x}")



def get_int():
    while True:
    try:
        x = int(input("what's x? "))
    except ValueError:
        print('x is not an integer')
    else:
        return x
```

```
main()
```

return can break the code too, we can get fancier too

```
def get_int():
    while True:
    try:
        return int(input("what's x? "))
    except ValueError:
        print('x is not an integer')
```

which is better? depends on what you believe

pass

its boring to tell the user its not an integer multiple times

we can pass

```
def get_int():
    while True:
    try:
        return int(input("what's x? "))
    except ValueError:
        pass
```

Remember that indentation in py is super important

function arguments

we can add a parameter for our function

```
def main():
    x = get_int('What's x? ')
    print(f'x is {x}')
```

```
def get_int(prompt):
    while True:
    try:
        return int(input(prompt))
    except ValueError:
        pass


main()
```

▼ 4. Libraries

modules

aka libraries that are built by me or others

lets see our first library

random

```
from random import choice

coin = choice(["heads", "tails"])
print(coin)
```

choice picks one from a list

we can import the whole library not just a function

```
import random

coin = random.choice(["heads", "tails"])
print(coin)
```

then we have to specify the package name first before using the choice function

if I am only using choice, its better to import it alone

```
import random

number = random.randint(1,10)
print(number)
```

This returns a number between a and b inclusively

```
import random

cards = ["jack", "queen", "king"]
random.shuffle(cards)

for card in cards:
    print(card)
```

this shuffles the list in place

statistics

```
import statistics
print(statistics.mean([100,90]))
```

sys

this allows me to write **command line argument**

```
python hello.py
```

I can pass arguments after file name

```
import sys

print("hello, my name is", sys.argv[1])
```

sys.argv has list of all arguments written

```
python name.py Ahmed
# hello, my name is Ahmed
```

`sys.argv[0]` has the file name

What if the user forgets to write the command line argument? we need to try it

```
import sys

try:
    print("hello, my name is", sys.argv[1])
except IndexError:
    print("Too few arguements")
```

```
python name.py # too few arguments
```

we can write it using if statements

```
import sys

if len(sys.argv)<2:
    print("Too few errors")
elif len(sys.argv)>2:
    print("too many arguements")
else:
    print("hello, my name is", sys.argv[1])
```

Notice how our code is berried inside else? its better to change this code

```
import sys

#check for errors
if len(sys.argv) < 2:
    print("Too few arguments")
```

```
elif len(sys.argv) > 2:
    print("Too many arguments")

#bug, this line will always work
print("hello, my name is", sys.argv[1])
```

which introduces a bug cuz the last line will always run, so we use `sys.exit` which closes our program

```
import sys

#check for errors
if len(sys.argv) < 2:
    sys.exit("Too few arguments")
elif len(sys.argv) > 2:
    sys.exit("Too many arguments")


print("hello, my name is", sys.argv[1])
```

this will work only if the code is correct

what if we have many arguments

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")

for arg in sys.argv:
    print('hello, my name is', arg)
```

```
python name.py David Carter Rongxin
```

which has an error cuz it prints name too, we need to **slice**

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")

for arg in sys.argv[1:]:
    print('hello, my name is', arg)
```

packages

aka third party libraries that people wrote already for me. found on pipy

install using `pip`

```
pip install cowsay
```

cowsay

```
import cowsay
import sys

if len(sys.argv) == 2:
    cowsay.cow("hello, " + sys.argv[1])
```

```
import cowsay
import sys

if len(sys.argv) == 2:
    cowsay.trex("hello, " + sys.argv[1])
```

```
python main.py Ahmed
```

requests

we have package that deals with API

```
pip install requests
```

itunes has its own API, lets deal with it

we can use this API to search for songs on their database

it returns **JSON**

```
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()

response = requests.get("https://itunes.apple.com/search?enti
    + sys.argv[1])
print(response.json())
```

```
python main.py weezer
```

This return json as dictionary

so we use the package `json` that can **format json**

```
import json
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()
response = requests.get(
"https://itunes.apple.com/search?entity=song&limit=1&term=" +
print(json.dumps(response.json(),indent=2))
```

This returns a dictionary, `resultCount` and `result`

result here is a list. this list contains a dictionary

so we have dictionary inside list inside dictionary

we care about `trackName`

we can iterate over this mess easily

```
import json
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()
response = requests.get(
"https://itunes.apple.com/search?entity=song&limit=50&term="

o = response.json()
for result in o['results']:
    print(result["trackName"])
```

This gets 50 songs by the song

custom libraries

we can have our own functions

```
# in a file called sayings.py

def main():
    hello("world")
    goodbye("world")

def hello(name):
    print(f"hello, {name}")
```

```
def goodbye(name):
    print(f"goodbye, {name}")

main()
```

Lets split the functionality into files

```
# say.py

import sys
from sayings import hello

if len(sys.argv) == 2:
    hello(sys.argv[1])
```

```
python say.py Ahmed
```

This has a problem, when I use sayings inside say, sayings prints stuff too,

lets fix it

```
# in a file called sayings.py

def main():
    hello("world")
    goodbye("world")

def hello(name):
    print(f"hello, {name}")

def goodbye(name):
    print(f"goodbye, {name}")
```

```
if __name__ == "main":
    main()
```

▼ 5. Unit Tests

testing calculator.py

Here is our calculator file

```
#calculator.py
def main():
    x = int(input("whats x? "))
    print("x squared is ", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
main()
```

we need to try corner case numbers, like zero, negatives, etc..

we use `__name__` so it runs only when I run calculator

Now lets test it

```
# test_calculator.py
from calculator import square

def main():
    test_square()

def test_square():
    if square(2) != 4:
        print("2 squared was not 4")
```

```
    if square(3) != 9:
        print("3 squared was not 9")


if __name__ == "__main__":
main()
```

we know that square should get 4, so if its not equal to 4, we have a problem

If in calculator we return plus instead of times, only square 3 will break

we need to test more

assert

notice that our function is two lines long, test is 5 lines, we need another way

```
# test_calculator.py
from calculator import square

def main():
    test_square()

def test_square():
    assert square(2) == 4
    assert square(3) == 9

if __name__ == "__main__":
    main()
```

This will throw an error, we need to handle it

```
# test_calculator.py
from calculator import square

def main():
```

```
    test_square()

def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print('2 squared was not 4')

    try:
        assert square(3) == 9
    except AssertionError:
        print('3 squared was not 9')

if __name__ == "__main__":
    main()
```

this looks horrible, we are testing two lines using dozens of lines

instead use a package `pytest`

pytest

admire the beauty

```
from calculator import square

def test_square():
    assert square(2) == 4
    assert square(-2) == 4
    assert square(3) == 9
    assert square(0) == 0
```

and in the terminal write this

```
pytest test_calculator.py
```

red output means I failed :(

Notice that the input is in main file, while square takes a value and returns a value. This is why

categories of tests

instead of having massive checks, its better to see if a chunk will fail

like maybe negative numbers fail, positive numbers pass

```
from calculator import square

def test_positive():
    assert square(2) == 4
    assert square(3) == 9

def test_zero():
    assert square(0) == 0

def test_negative():
    assert square(-2) == 4
    assert square(-3) == 9
```

This helps me know what fails.

testing for exceptions

square takes int, not strings

```
import pytest
from calculator import square

def test_positive():
    assert square(2) == 4
    assert square(3) == 9

def test_zero():
```

```
    assert square(0) == 0

def test_negative():
    assert square(-2) == 4
    assert square(-3) == 9

def test_str():
    with pytest.raises(TypeError):
        square('cat')
```

side effects and testing

```
# hello.py
def main():
    name = input("whats your name? ")
    hello(name)

def hello(to="world"):
    # side effect
    print("hello,", to)

if __name__ == "__main__":
    main()
```

notice that hello prints not return, aka its a side effect

```
# test_hello.py
from hello import hello

def test_hello():
    assert hello("David") == "hello, David"
```

will this work? Nope, cut it returns nothing

instead, we want our function to return the string not print

```
def main():
    name = input("whats your name? ")
    print(hello(name))

def hello(to="world"):
    return f"hello, {to}"

if __name__ == "__main__":
    main()
```

now it works

```
#test_hello.py
from hello import hello

def test_default():
    assert hello() == "hello, world"

def test_argument():
    assert hello("David") == "hello, David"
```

Note, we can test using loops, but its better for the tests to be super easy

```
#test_hello.py
from hello import hello

def test_default():
    assert hello() == "hello, world"

def test_argument():
    for name in ["Hermione","Harry","Ron"]:
        assert hello(name) == f"hello, {name}"
```

collection of tests

its better to have a folder for tests

```
mkdir test
code test/test_hello.py
code test/__init__.py
```

```
#test_hello.py
from hello import hello

def test_default():
    assert hello() == "hello, world"

def test_argument():
    assert hello("David") == "hello, David"
```

we need to have code test/__init__.py so our folder is treated like **package** and can be tested

```
pytest test
```

now our pytest will find every test in the folder

▼ 6. File I/O

list

we can save many stuff into lists

```
#names.py
names = []

for _ in range(3):
    names.append(input("whats your name? "))

for name in sorted(names):
    print(f"hello, {name}")
```

this asks for three names, then prints them sorted

Unfortunately, when I close my program, I lose my list

I want to save it

open

```
name = input("whats your name? ")

# file will be rewritten each time
file = open("names.txt", "w")
file.write(name)
file.close()
```

if names . txt does not exist, it will be made, it will be rewritten every time I use my code.

opening is like double clicking, If I don’t close, it will keep taking the resources

to append the code instead of rewriting

```
name = input("whats your name? ")

file = open("names.txt", "a")
file.write(name)
file.close()
```

This has a problem. they are written sticking to each other, i want to add new line

```
name = input("whats your name? ")

file = open("names.txt", "a")
file.write(f'{name}\n')
file.close()
```

with

instead of forgetting to close the file, we write it using with

```
name = input("whats your name? ")

with open("names.txt", "a") as file:
file.write(f"{name}\n")
```

This is better, but how to read my file?

```
with open("names.txt", "r") as file:
    lines = file.readlines()

for line in lines:
    print("hello,", line.rstrip())
```

remember that we had \n at the end of the line, rstrip will remove it

there is a **better way**

```
with open("names.txt", "r") as file:
    for line in file:
        print("hello,", line.rstrip())
```

sorted

```
names = []
with open("names.txt") as file:
    for line in file:
        names.append(line.rstrip())

for name in sorted(names):
    print(f"hello, {name}")
```

to sort, we created a list, appended it to list, then manipulated the list

a better way

```
with open("names.txt") as file:
    for line in sorted(file):
        print("hello", line.rstrip())
```

we can run sorted on the file itself

comma separated values

csv, better than txt. cuz I can store both names and houses in csv, as two columns separated by comma

```
# students.csv
Hermione,Gryffindor
Harry,Gryffindor
Ron,Gryffindor
Draco,Slytherin
```

```
with open("students.csv") as file:
    for line in file:
        row = line.rstrip().split(",")
        print(f"{row[0]} is in {row[1]}")
```

we can print names and houses by splitting the row by comma and indexing

we can get fancier

```
with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        print(f"{name} is in {house}")
```

we can save name, house in the same line

its better to save the results in my list

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        students.append(f"{name} is in {house}")

for student in sorted(students):
    print(student)
```

but its better to store it as dictionary inside list

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {}
        student["name"] = name
        student["house"] = house
        students.append(student)

for student in students:
    print(f"{student['name']} is in {student['house']}"
```

we can save the dictionary in one line instead of two

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {"name":name, "house":house}
        students.append(student)
```

```
for student in students:
    print(f"{student['name']} is in {student['house']}")
```

But how to sort them? we can use key

sort keys

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {"name":name, "house":house}
        students.append(student)

def get_name(student):
    return student['name']

for student in sorted(students, key = get_name):
    print(f"{student['name']} is in {student['house']}")
```

we have a function that just returns the name, we use it as a key for sorted

we can reverse them too or sort with house

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {"name":name, "house":house}
        students.append(student)

def get_house(student):
    return student['house']
```

```
for student in sorted(students, key = get_house, reverse = Tr
    print(f"{student['name']} is in {student['house']}")
```

lambda function

Do I really to make a whole function to use it as a key??

I can just use anonymous functions

```
students = []

with open("students.csv") as file:
    for line in file:
        name, house = line.rstrip().split(",")
        student = {"name":name, "house":house}
        students.append(student)


for student in sorted(students, key = lambda student: student
    print(f"{student['name']} is in {student['house']}")
```

Remember, if we enter house as Number four, privet drive

it will break the code cuz it has csv

we can think of smart solutions, but the best, use package

csv library

```
import csv

students = []

with open("students.csv") as file:
    reader = csv.reader(file)
    for name,home in reader:
        students.append({"name":name, "home":home})
```

```
for student in sorted(students, key=lambda student: student["
    print(f"{student['name']} is in {student['home']}")
```

Now, it can be read without breaking the code

csv DictReader

we can make our csv have column names

then we use dictreader

```
import csv
students = []

with open("students.csv") as file:
    reader = csv.DictReader(file)
    for row in reader:
        students.append({"name": row["name"], "home": row["ho

for student in sorted(students, key=lambda student: student["
    print(f"{student['name']} is in {student['home']}")
```

now, if our csv is house-name instead of name-house, it will not break

csv DictWriter

if our csv has name and house column names only, we can fill it

```
import csv
name = input("what's your name? ")
home = input("where's yout home? ")

with open("students.csv", "a") as file:
    writer = csv.writer(file)
    writer.writerow([name,home])
```

we can have a better way using dict writer

```
import csv
name = input("what's your name? ")
home = input("where's yout home? ")

with open("students.csv", "a") as file:
    writer = csv.DictWriter(file, fieldnames=["name","home"])
    writer.writerow({"name": name, "home": home})
```

Images, PIL library

pillow library allows us deal with images

we can create gifs from images by looping

```
import sys
from PIL import Image

images = []

for arg in sys.argv:
    image = Image.open(arg)
    images.append(image)

images[0].save(
    'costumes.gif', save_all = True, append_images = [images[
    duration = 200, loop = 0
)
```

▼ 7. Regular Expressions

pattern of mystery code to validate input and search

```
# validate.py
email = input("what's your email? ").strip()
```

```
if "@" in email:
    print("Valid")
else:
    print("Invalid")
```

we use `in` to see if the @ is in the email, to make sure its a real email

but if i just input @ it will work

maybe look for @ and .?

```
# validate.py
email = input("what's your email? ").strip()

if "@" in email and '.' in email:
    print("Valid")
else:
    print("Invalid")
```

still not enough, maybe we can split and check for username

```
email = input("what's your email? ").strip()
username, domain = email.split("@")

if (username) and ("." in domain):
    print("Valid")
else:
    print("Invalid")
```

if username exists, it will be true, this is called truthy

now to test that the person has domain edu

```
email = input("what's your email? ").strip()
username, domain = email.split("@")

if username and domain.endswith('.edu'):
```

```
    print("Valid")
else:
    print("Invalid")
```

this is escalating quickly

regular expression pattern

```
import re

email = input("what's your email? ").strip()

if re.search("@", email):
    print("Valid")
else:
    print("Invalid")
```

this is like the first version, searches for @ in the email.

we want text then @ then text

patterns

```
. # any character except a newline
* # 0 or more repetitions
+ # 1 or more repetitions
? # 0 or 1 repetitions
{m} # m repetitions
{m,n} # m-n repetitions
```

we will explain this witchcraft as we go

we want **multiple text** before @ and **after**

```
import re

email = input("what's your email? ").strip()
```

```
if re.search((".*@.*", email):
    print("Valid")
else:
    print("Invalid")
```

```
malan@
```

our code is wrong cuz * accepts 0 or more, use + instead cuz it uses 1 or more characters

```
import re

email = input("what's your email? ").strip()

if re.search((".+@.+", email):
    print("Valid")
else:
    print("Invalid")
```

remember, we can use many stuff to solve the same problem

we can say we have a **character then 0 or more characters**

```
import re

email = input("what's your email? ").strip()

if re.search(("..*@..*", email):
    print("Valid")
else:
    print("Invalid")
```

Now we want out email to end with .edu. to print the . we use escape characters

```
import re
email = input("what's your email? ").strip()
```

```
if re.search(r".+@.+\.edu", email):
    print("Valid")
else:
    print("Invalid")
```

we use r before the string to tell py not to interpret the escape characters the normal way

matching start and end

search such checks if our input has the email or not

so we need new keywords

```
^ # matches the start of the string
$ # matches the end of the string or
    # just before the newline at the end of the string
```

so we add ^ and $ to make the input consists of email only

literally: start with email, end with email

```
import re
email = input("what's your email? ").strip()

if re.search(r"^.+@.+\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

sets of characters

```
malan@@@harvard.edu
```

we need new syntax

```
[] # set of characters
[^] # complementing the set
```

[^] means the characters can be anything except the ones written inside

```
import re
email = input("what's your email? ").strip()

if re.search(r"^[^@]+@[^@]+\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

lets read it slowly:

match from the start and the end

[^@] have any character except @

then + means one or more repetition

Actually, we need to allow all letters and number

```
- # range
```

```
import re
email = input("what's your email? ").strip()

if re.search(r"^[a-zA-z0-9_]+@[a-zA-z0-9_]+\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

we now allow for all numbers, letters small case and upper, and _

since this is so common, we have a shortcut for it `\w`

character classes

```
import re
email = input("what's your email? ").strip()

if re.search(r"^\w+@\w+\.edu$", email):
    print("Valid")
else:
    print("Invalid")
```

Here are other character classes

```
\d # decimal digit
\D # not a decimal digit
\s # whitespace charcters
\S # not a whitespace character
\w # word character...as well as numbers and the underscore
\W # not a word character
```

decimal digit mean numbers

```
A|B # either A or B
(...) # a group
(?:...) # non-capturing version
```

we usually combine () with A|B. More on them after a while

```
(\w|\s)
```

means words or numbers or underscore or space

Notice: till now, we don’t capture uppercase

```
MALAN@HARVARD.EDU
```

returns invalid

Flags

we can ignore case, or handle multiple lines

```
re.IGNORECASE
re.MULTILINE
re.DOTALL
```

```
import re
email = input("what's your email? ").strip()

if re.search(r"^\w+@\w+\.edu$", email, re.IGNORECASE):
    print("Valid")
else:
    print("Invalid")
```

Now we have a problem

```
malan@cs50.harvard.edu
```

returns invalid

groups

```
A|B # either A or B
(...) # a group
(?:...) # non-capturing version
```

lets use groups

```
import re
email = input("what's your email? ").strip()

if re.search(r"^\w+@(\w+\.)?\w+\.edu$",email,re.IGNORECASE):
    print("Valid")
```

```
else:
    print("Invalid")
```

(\w+\.)? is saying there is zero or more words aka its optional

email address

we were close, here is the full version

```
^[a-zA-Z0-9.!#$%&'*+\/=?^_`
{|}~-]+@[a-zA-Z0-9](?:[a-zA
-Z0-9-]{0,61}[a-zA-Z0-9])?(
?:\.[a-zA-Z0-9](?:[a-zA-Z0-
9-]{0,61}[a-zA-Z0-9])?)*$
```

although cryptic, unfortunately, we can read it now

Its better to use libraries

match

```
re.fullmatch # searches from beginning to end
re.match # searches from beginning
```

so no need to specify the ^

clean user input

we can format the input into what we expect

```
name = input("what's your name? ").strip()
print(f"hello, {name}")
```

I can write David Malan or Malan, David

How to fix this?

```
name = input("what's your name? ").strip()

if ',' in nameL
    last, first = name.split(', ')
    name = f'{first} {last}'

print(f"hello, {name}")
```

notice the split splits on comma and space

we want the space to optional, we need regex

```
import re
name = input("what's your name? ").strip()

matches = re.search(r"^(.+), *(.+)$", name)

if matches:
    name = matches.group(2) + " " + matches.group(1)

print(f"hello, {name}")
```

we use () grouping to capture aka store the result in matches

when we return the groups, we can store them using `matches.groups`

or `match.group(i)`

```
David Malan
Malan, David
Malan,David
Malan,      David
```

all here return valid

Note here: first group is in group(1) not zero

walrus operator

we can define match and check if it exists or not at the same time

using `:=`

```
import re
name = input("what's your name? ").strip()

if matches := re.search(r"^(.+), *(.+)$", name):
    name = matches.group(2) + " " + matches.group(1)

print(f"hello, {name}")
```

re.sub

lets deal with url now

```
url = input("URL: ").strip()

username = url.replace("https://twitter.com/", "")
print(f"Username: {username}")
```

to capture username, just throw away the link part by replacing it

but it will fail if he enters http or slash after username, or www

a better way is to `removeprefix`

```
url = input("URL: ").strip()

username = url.removeprefix("https://twitter.com/")
print(f"Username: {username}")
```

To get fancy, we need regex

`re.sub` means substitution

```
import re
url = input("URL: ").strip()

username = re.sub(r"https://twitter\.com/", "", url)
print(f"username: {username}")
```

this replaces the link part with nothing

```
import re
url = input("URL: ").strip()

username = re.sub(r"^(https?://)?(www\.)?twitter\.com/",
         "", url)
print(f"username: {username}")
```

we can make them optional, and s in https optional

Notice how s in https is optional inside the optional https

Lesson to take: take baby steps

we need logic to make sure the user enters url for twitter

```
import re

url = input("URL: ").strip()
print(url)

matches = re.search(
r"^https?://(www\.)?twitter\.com/(.+)$", url, re.IGNORECASE)

if matches:
    print(f"Username:", matches.group(2))
```

remember: to make www optional, we used ()?, but we did capture it

lets tidy it more and use the **non capturing group**

```
import re

url = input("URL: ").strip()
print(url)

if matches := re.search(
r"^https?://(?:www\.)?twitter\.com/(.+)$",
url,re.IGNORECASE):
    print(f"Username:", matches.group(2))
```

tip: visit https://regex101.com/

we also have `split` and `findall`

▼ 8. OOP

Here we are :)

tuple

we have procedural programming, functional programming and OOP

first lets see procedural

will it have problems as code gets more complex?

```
name = input("Name: ")
house = input("House: ")

print(f"{name} from {house}")
```

Lets turn it into a function

```
def main():
    name = get_name()
    house = get_house()
    print(f"{name} from {house}")
```

```
def get_name():
    return input("Name: ")

def get_house():
    return input("House: ")

if __name__ == "__main__":
    main()
```

Our code is still working,

it will be better to get student that returns both name and house

we can use **tuple**

```
def main():
        name, house = get_student()
    print(f"{name} from {house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return name, house

if __name__ == "__main__":
    main()
```

What we are actually doing is using tuple

tuple is like list but can't be modified

`name, house` is a tuple

but its better to use the brackets

```
def main():
        student = get_student()
    print(f"{student[0]} from {student[1]}")
```

```
def get_student():
    name = input("Name: ")
    house = input("House: ")
    return (name, house)

if __name__ == "__main__":
    main()
```

Remember that tuples are immutable, can’t be modified

```
def main():
        student = get_student()
        if student[0] == 'Padma':
            student[1] = 'Racenclaw'
    print(f"{student[0]} from {student[1]}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return (name, house)

if __name__ == "__main__":
    main()
```

This will result an error, to fix it. use lists instead

```
def main():
        student = get_student()
        if student[0] == 'Padma':
            student[1] = 'Racenclaw'
    print(f"{student[0]} from {student[1]}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
```

```
    return [name, house]

if __name__ == "__main__":
    main()
```

dictionary

dict will be good here too, so we don’t confuse the order of the indexing

```
def main():
    student = get_student()
    print(f"{student['name']} from {student['house']}")

def get_student():
    student = {}
    student["name"] = input("Name: ")
    student["house"] = input("House: ")
return student

if __name__ == "__main__":
    main()
```

now, no need to remember the indexing, we write name and house.

lets get fancier

```
def main():
    student = get_student()
    if student["name"] == "padma":
        student["house"] = "ravenclaw"
    print(f"{student['name']} from {student['house']}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
```

```
    return {"name": name, "house": house}
if __name__ == "__main__":
main()
```

dictionaries are mutable, notice how get_student now creates a dictionary on the fly

classes and objects

dictionary is powerful, but it would have been better if python had a data structure called student

Instead, we have `class` which is a blueprint that allows us to do own own objects

```
class Student:
    ...

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    student = Student()
    # attribute
    student.name = input('Name: ')
    student.house = input('House: ')
    return student

if __name__ == "__main__":
main()
```

classes are always capitalized, … means pass

think of `attribute` as **variables** related to the object

student = Student(): student is the object, Student is the class

we say: student is an **instance** of the Student

classes can be mutable or immutable, depending on my implementation

instance method

Its better to pass the names inside our class

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    name = input('Name: ')
    house = input('House: ')
    student = Student(name, house)
    return student

if __name__ == "__main__":
main()
```

why this is better? I now have more control of the variables, i can error check and other stuff

classes have attributes and methods

think of methods as functions related to the object

 think of self as h marbota in Arabic

`init` method creates the variables when we create the object aka the attribute

Lets clean the code a little bit

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    name = input('Name: ')
    house = input('House: ')
    return Student(name, house)


if __name__ == "__main__":
main()
```

validating attribute

OOP encourages us to do everything related to the class inside the class not outside

so we can validate the attributes (variables) inside the class

if we don't receive a name, using `sys.exit` is really extreme

using `return none` means I already created the student

instead, use `raise`

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError('Missing name')
        self.name = name
        self.house = house
```

```
def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    name = input('Name: ')
    house = input('House: ')
    try:
        return Student(name, house)
    else:
        pass


if __name__ == "__main__":
main()
```

we will not focus on the try else part now so I will remove it

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
        self.name = name
        self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    name = input("Name: ")
```

```
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

string method

It will be better to just print student and get name and house

if we try it now

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
        self.name = name
        self.house = house

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

I will get that its object stored on memory.

Solution: use `str`

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
        self.name = name
        self.house = house

    def __str__(self):
        return 'a student'

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

now it will print a student, we can make use of self

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
        self.name = name
```

```
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

str has a similar method, str is for the user

custom methods

lets add patronous

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff", "ravencl
            raise ValueError("invalid house")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"
```

```
def main():
    student = get_student()
    print(student)


def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus = input("Patrnous: ")
    return Student(name, house, patronus)


if __name__ == "__main__":
    main()
```

we can create our own method aka function, like casting the Patronus

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff", "ravencl
            raise ValueError("invalid house")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"

    def charm(self):
        match self.patronus:
            case 'Stag':
                return '🦌'
            case 'otter':
```

```
                return '🦦'
            case 'Jack Russel terrier':
                return '🐶'
            case _:
                return '✨'


def main():
    student = get_student()
    print('Expecto Patronum!')
    print(student.charm())
    print(student)


def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus = input("Patrnous: ")
    return Student(name, house, patronus)


if __name__ == "__main__":
    main()
```

properties, getters and setters

Lets clean our code to focus on the new stuff

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
```

```
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

Although we are validating our code, it is bad, why?

in main function, I can change the house after all of this trouble

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

def main():
```

```
    student = get_student()
    student.house = 'baka'
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

solution? use `properties`

properties are attributes that we have more control over them

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
            raise ValueError("invalid house")
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

    # getter
    def house(self):
        return self.house

    # setter
    def house(self, house):
        self.house = house
```

```
def main():
    student = get_student()
    student.house = 'baka'
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

this code has mistake, but for now, getter gets the value of the attribute

setter changes the value of the attribute

we will move our error checking inside the setter function, why?

if the user tries to modify the value using

```
student.house = 'baka'
```

the setter function will be run magically

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

    # getter
    @property
```

```
    def house(self):
        return self.house

    # setter
    @house.setter
    def house(self, house):
            if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
                raise ValueError("invalid house")
        self.house = house

def main():
    student = get_student()
    student.house = 'baka'
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

property name must be equal to variable name

note: we can now remove the validation from the init method

running self. house = house runs the setter

note: we can’t have property name same as instance name

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("missing name")
        self.name = name
        self.house = house
```

```
    def __str__(self):
        return f'{self.name} from {self.house}'

    # getter
    @property
    def house(self):
        return self._house

    # setter
    @house.setter
    def house(self, house):
            if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
                raise ValueError("invalid house")
        self._house = house

def main():
    student = get_student()
    student.house = 'baka'
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

now baka will break the code now thanks to the validation

lets do a property for name too

```
class Student:
    def __init__(self, name, house):
```

```
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        if not name:
            raise ValueError("missing name")
        self._name = name

    @property
    def house(self):
        return self._house

    @house.setter
    def house(self, house):
            if house not in ["gryffindor", "hufflepuff",
         "ravenclaw", "slytherin"]:
                raise ValueError("invalid house")
        self._house = house

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)
```

```
if __name__ == "__main__":
    main()
```

now here is why py is baka

if I write

```
student._house = 'baka'
```

it will be changed :)

py depends on honor system, if you see _ please don’t touch it

types and classes

we were already using classes all this time

int is an object

str is an object

list is an object, dict is an object

proof

```
print(type(50))
print(type('hello'))
print(type(list()))
print(type(dict()))
```

class methods

sometimes, we want functionality to be with the class not the object

every object will have the same functionality

lets do the sorting hat from harry potter

```
class Hat:
    ...
```

```
hat = Hat()
```

now we instantiated the object hat. we want the functionality to sort students to houses

```
class Hat:
    def sort(self, name):
        print(name, 'is in', 'some house')

hat = Hat()
hat.sort("Harry")
```

now create the houses

```
import random

class Hat:
    def __init__(self)
        self.houses = ["gryffindor", "hufflepuff", "ravenclaw

    def sort(self, name):
        print(name, 'is in', random.choice(self.houses))

hat = Hat()
hat.sort("Harry")
```

its useful to have houses in the init so we can use it multiple functions later on or adjust it

we use class to represent **real world entity**

But we are not using classes the right way here

There is only one sorting hat, I just want one hat

this is why we have class methods

```
import random

class Hat:
    houses = ["gryffindor", "hufflepuff", "ravenclaw", "slyth

    @classmethod
    def sort(cls, name):
        print(name, "is in", random.choice(cls.houses))

Hat.sort("Harry")
```

we use cls instead of self

we remove init cuz we do not instantiate

houses becomes a class variable

sort is a class method

so we just use Hat, no hat = Hat()

we can also modify the student code again

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'


def main():
    student = get_student()
    print(student)

def get_student():
```

```
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__":
    main()
```

why do I have a function called get_student that is related to a student class but outside it?? it should be inside it

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

    def __str__(self):
        return f'{self.name} from {self.house}'

    @classmethod
    def get(cls):
            name = input("Name: ")
            house = input("House: ")
            return cls(name, house)


def main():
    student = Student.get()
    print(student)


if __name__ == "__main__":
    main()
```

class method means we can use it without instantiating

when we get the results, we can use them to instantiate the object

🚀 classes are at the top of the file or separate files

There are also static methods but ignore it for now

inheritance

we can have many classes that they inherit from each other

like student and professor

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

class Professor:
    def __init__(self,name,subject):
        self.name = name
        self.subject = subject
```

Both have name, imagine validating name in both.

This is a red flag

they are both wizards, so have a wizard to inherit from

inherit using `super`

```
class Wizard:
    def __init__(self,name):
        if not name:
            raise ValueError()
        self.name = name

class Student(Wizard):
    def __init__(self, name, house):
        super().__init__(name)
```

```
        self.house = house

class Professor(Wizard):
    def __init__(self, name, subject):
        super().__init__(name)
        self.subject = subject

wizard = Wizard("Albus")
student = Student("Harry","Gryffindor")
print(student.name)
```

Note: exceptions in python actually inherit from parent called `Exception` that inherits from `BaseException`

Operator overloading

we can make + to add, or to concatenate

aka its **overloaded**

```
class Vault:
    def __init__(self,galleons=0,sickles=0,knuts=0):
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f" {self.galleons}, {self.sickles}, {self.knut


potter = Vault(100,50,25)
print(potter)

weasley = Vault(25,50,100)
print(weasley)
```

How to combine potter money and weasley galleons?

```
galleons = potter.galleons + weasley.galleons
total = Vault(galleons)
print(total)
```

which is ok, but it would be better if I can just say potter + weasley

aka overload

```
class Vault:
    def __init__(self, galleons=0, sickles=0, knuts=0):
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f"{self.galleons} Galleons, {self.sickles} Sic

    def __add__(self, other):
        galleons = self.galleons + other.galleons
        sickles = self.sickles + other.sickles
        knuts = self.knuts + other.knuts
        return Vault(galleons, sickles, knuts)


potter = Vault(100, 50, 25)
print(potter)

weasley = Vault(25, 50, 100)
print(weasley)

total = potter + weasley
print(total)
```

when dealing with two objects, refer to the other object as `other`

potter + weasly:- self + other

we return a vault.

▼ 9. Et Cetera

This lecture covers extra stuff that we can do beyond the fundamentals

set

a list that has **no duplicates**

```
students = [
{'name': "Hermoine", 'house': 'Gryffindor'},
{'name': "Harry", 'house': 'Gryffindor'},
{'name': "Ron", 'house': 'Gryffindor'},
{'name': "Draco", 'house': 'Slytherin'},
{'name': "Padma", 'house': 'Ravenclaw'}
]
```

Problem: how to remove duplicates?

we can do it using lists

```
houses = []
for student in students:
    if student['house'] not in houses:
        houses.append(student['house'])

for house in sorted(houses):
    print(house)
```

but this is like reinventing the wheel

instead, use set

```
houses = set()
for student in students:
    houses.add(student['house'])
```

```
for house in sorted(houses):
    print(house)
```

global

global variable, is a variable that is outside functions

lets create `bank.py`

```
balance = 0

def main():
    print("Balance:", balance)


if __name__ == "__main__"
    main()
```

this prints balance: 0

 I can read a variable outside the functions

lets modify the balance

```
balance = 0

def main():
    print("Balance:", balance)
    deposit(100)
    withdraw(50)
    print("Balance:", balance)


def deposit(n):
```

```
    balance += n

def withdraw(n):
    balance -= n


if __name__ == "__main__"
    main()
```

we get an error `UnboundedLocalError`

 can't modify a variable defined outside functions from within functions

lets try to solve it

```
def main():
    balance = 0
    print("Balance:", balance)
    deposit(100)
    withdraw(50)
    print("Balance:", balance)


def deposit(n):
    balance += n

def withdraw(n):
    balance -= n


if __name__ == "__main__"
    main()
```

still error, balance is local to main only

solution: use `global`

```
balance = 0

def main():
    print("Balance:", balance)
    deposit(100)
    withdraw(50)
    print("Balance:", balance)


def deposit(n):
    global balance
    balance += n

def withdraw(n):
    global balance
    balance -= n


if __name__ == "__main__"
    main()
```

A better solution is to use OOP due to the encapsulation

```
class Account:
    def __init__(self):
        self._balance = 0

    @property
    def balance(self):
        return self._balance

    def deposit(self, n):
        self._balance += n
```

```
    def withdraw(self, n):
         self._balance -= n

def main():
    account = Account()
    print('Balance:', account.balance)
    account.deposit(100)
    account.withdraw(50)
    print('Balance:', account.balance)

main()
```

This is better, because attributes are designed to pass to all methods

use global with super small programs not modules.

Remember: `@property` is like a setter

Constants

meow.py

```
for _ in range(3):
    print('meow')
```

instead of hardcoding 3, if it won't change, write it as a constant

but py does not have constants, so write it uppercase

```
MEOWS = 3
for _ in range(3):
    print('meow')
```

Notice: I can still modify meows

we can also have class constants

```
class Cat:
    MEOWS = 3
```

```
    def meow(self):
        for _ in range(Cat.MEOWS):
            print('meow')

cat = Cat()
cat.meow()
```

type hints `mypy`

py figures out what type is right for the variable, unlike other programming languages

we can hint the user to use specific type though

using the package `mypy`

```
def meow(n):
    for _ in range(n):
        print('meow')

number = input('Number: ')
meow(number)
```

notice our mistake, we need to convert numbers to int

to hint: use `: int`

```
def meow(n: int):
    for _ in range(n):
        print('meow')

number = input('Number: ')
meow(number)
```

it should be int. not a must

we can use `mypy` to check if the types are as specified or not

in the terminal

```
mypy meows.py
```

to help mypy, we can tell it what variables types should be too

```
def meow(n: int):
    for _ in range(n):
        print('meow')

number: int = int(input('Number: '))
meow(number)
```

Its good practice, even though its not enforced by python

```
def meow(n: int):
    for _ in range(n):
        print('meow')

number: int = int(input('Number: '))
meows: str = meow(number)
print(meows)
```

remember that meow just does a side effect and returns none

so meows is actually none not str

to specify that our function returns none

use `-> None`

```
def meow(n: int) -> None:
    for _ in range(n):
        print('meow')

number: int = int(input('Number: '))
```

```
meows: str = meow(number)
print(meows)
```

now `mypy` catches the str error

to fix it

```
def meow(n: int) -> str:
    reutrn 'meow\n' * n

number: int = int(input('Number: '))
meows: str = meow(number)
print(meows, end = '')
```

problem fixed

docstrings

This is how we document our function using `'''`

```
def meow(n: int) -> str:
    """Meow n times."""
    reutrn 'meow\n' * n

number: int = int(input('Number: '))
meows: str = meow(number)
print(meows, end = '')
```

There are packages that can turn this document into pdfs.

There are other documenting ways

```
def meow(n: int) -> str:
    """
    Meow n times.

    :param n: Number of times to meow
```

```
    :type n: int
    :raise TypeError: If n is not an int
    :return: A string of n meos, one per line
    :rtype: str
    """
    reutrn 'meow\n' * n

number: int = int(input('Number: '))
meows: str = meow(number)
print(meows, end = '')
```

This is just a way to document so it can be easy to turn into pdf

argparse

what if we take input from the terminal

```
import sys

if len(sys.argv) == 1:
    print('meow')
else:
    print('usage: meows.py')
```

Notice: when I write

```
python meows.py
```

args = 1

But a better way is to be able to use flags: `-n`

```
import sys

if len(sys.argv) == 1:
    print('meow')
```

```
elif len(sys.argv) == 3 and sys.argv[1] == '-n':
    n = int(sys.argv[2])
    for _ in range(n):
        print('meow')

else:
    print('usage: meows.py')
```

```
python meows.py -n 3
```

to do other flags, it will get super hard, so we use `argparse`

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-n')
args = parser.parse_args()

for _ in range(int(args.n)):
    print('meow')
```

```
python meows.py -n 3
```

lets add the help `-h`

```
python meows.py -h
```

to make it more helpful

```
import argparse

parser = argparse.ArgumentParser(description = 'meow like a c
parser.add_argument('-n', help = 'number pf times to meow',
                                        type = int, default = 1)
```

```
args = parser.parse_args()

for _ in range(args.n):
    print('meow')
```

unpacking

```
first, _ = input('what's your name').split(' ')
print(f'hello, {first}')
```

we name last as `_` cuz its not important

we unpack here by splitting the input into two variables

but there are other ways

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

print(total(100,50,25), 'knuts')
```

this function tells galleons, sickles into knuts

lets store them into a list

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

coins = [100,50,25]

print(total(coins[0], coins[1], coins[2]), 'knuts')
```

which works but verbose, a better way is to **unpack** the list

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts
```

```
coins = [100,50,25]

print(total(*coins), 'knuts')
```

`*coins` gets out the coins

Notice, it will break if coins length increased

lets use parameters

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts


print(total(galleons = 100, sickles = 50, knuts = 25), 'knuts
```

this reminds us of dictionary, keys and values

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

coins = {'galleons' = 100, 'sickles' = 50, 'knuts' = 25}

print(total(**coins),'knuts')
```

`**coins` unpacks keys and values

*args, **kwargs

it can also be used when functions have unlimited number of inputs

```
def f(*args, **kwargs):
    print('positional:', args)

f(100,50,25)
f(100, 50, 25, 5)
f(100)
```

args are positional arguments, kwargs are named arguments

The above code returns a **tuple**

lets deal with named arguments too

this returns a **dictionary**

```
def f(*args, **kwargs):
    print('Named:', kwargs)

f(galleons = 100, sickles = 50, knuts = 25)
```

Remember print? it takes a variable number of inputs

```
print('hello','world')
```

map

python is really object oriented.

there is a third paradigm though, **functional programming**

```
# yell.py
def main():
    yell('this is cs50')

def yell(phrase):
    print(phrase.upper())

main()
```

its better to be a list

```
def main():
    yell(['this', 'is' ,'cs50'])

def yell(words):
```

```
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(*uppercased)

main()
```

works, but bad, why force me to use a list?

```
def main():
    yell('this', 'is' ,'cs50')

def yell(*words):
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(*uppercased)

main()
```

unpacking solved the problem, but there is even better

`map`

```
def main():
    yell('this', 'is' ,'cs50')

def yell(*words):
    uppercased = map(str.upper, words)
    print(*uppercased)

main()
```

list comprehension

this is the best way aka pythonic way

```
def main():
    yell('this', 'is' ,'cs50')

def yell(*words):
    uppercased = [word.upper() for word in words]
    print(*uppercased)

main()
```

filter

```
students = [
{'name': "Hermoine", 'house': 'Gryffindor'},
{'name': "Harry", 'house': 'Gryffindor'},
{'name': "Ron", 'house': 'Gryffindor'},
{'name': "Draco", 'house': 'Slytherin'},
{'name': "Padma", 'house': 'Ravenclaw'}
]

gryffindors = [
    student['name'] for student in students if student['house
]

for gryffindor in sorted(gryffindors):
    print(gryffindor)
```

a better way

```
students = [
{'name': "Hermoine", 'house': 'Gryffindor'},
{'name': "Harry", 'house': 'Gryffindor'},
{'name': "Ron", 'house': 'Gryffindor'},
{'name': "Draco", 'house': 'Slytherin'},
{'name': "Padma", 'house': 'Ravenclaw'}
```

```
]

def is_gryffindor(s):
    return s['house'] == 'Gryffindor'

gryffindors = filter(is_gryffindor, students)

for gryffindor in sorted(gryffindors, key=lambda s: s['name']
    print(gryffindor['name'])
```

filter expects a function that returns True and False.

dictionary comprehensions

```
students = ['Hermione', 'Harry', 'Ron']

gryffingdors = []

for student in students:
    gryffindors.append({'name': student, 'house':'Gryffindor'
```

This builds up the dictionary of students

here is a better way

```
students = ['Hermione', 'Harry', 'Ron']

gryffindors = [{'name': student, 'house': 'Gryffindor'} for s
```

This returns a list of dictionaries

what if we just want a dictionary

```
students = ['Hermione', 'Harry', 'Ron']

gryffindors = {student: "Gryffindor" for student in students}
```

enumerate

to order the students starting from 1

```
students = ['Hermione', 'Harry', 'Ron']

for i in range(len(students)):
    print(i + 1, students[i])
```

a better way is to use enumerate

```
students = ['Hermione', 'Harry', 'Ron']

for i, student in enumrate(students):
    print(i + 1, student)
```

generators, yield, iterators

When we are trying to sleep, imagine sheep that jump over a fence

```
n = int(input("what's n? "))
for i in range(n):
    print('🐑' * i)
```

lets make it as a function

```
def main():
    n = int(input("what's n? "))
    for i in range(n):
        print('🐑' * i)

main()
```

lets split it into functionality

```
def main():
    n = int(input("what's n? "))
    for i in range(n):
        print(sheep(i))

def sheep(n):
    return '🐑' * n

main()
```

get super fancier

```
def main():
        n = int(input("what's n? "))
        for s in sheep(n):
            print(s)

def sheep(n):
    flock = []
    for i in range(n):
        flock.append('🐑' * i)
    return flock
```

Now if we try to print 1,000,000 sheep. It will stop printing sheep

why? cuz I am trying to print everything at once,

instead, make the function return chunks of the solution by `generators`

```
def main():
        n = int(input("what's n? "))
        for s in sheep(n):
            print(s)

def sheep(n):
```

```
    for i in range(n):
        yield '🐑' * i
```

unlike break, yield does not break the code

This was CS50🤍