Why Python?

Here at Boot.dev we try to choose the languages we use in our courses wisely. Python isn't just a great learning tool. In fact, you'll likely use Python in a real-world job in your career.

Simple is better than complex

Python code is *simple*. Compare the way three different modern coding languages print text to the console.

Golang

```
fmt.Println("hey there")
```

JavaScript

```
console.log('hey there')
```

Python

```
print("hey there")
```

Assignment

Print `hello world` to the console.

Why Python?

Here are some reasons we think Python is a future-proof choice for developers:

- Easy to read and write - Python reads like plain English. Due to its simple syntax, its a great choice for implementing advanced concepts like AI. This is arguably Python's *best feature*.
- Popular - According to the Stack Overflow Developer Survey, Python is the 4th most popular coding language in 2020.
- Free - Python, like many languages nowadays, is developed under an open-source license. It's free to install, use, and distribute.
- Portable - Python written for one platform will work on any other platform.
- Interpreted - Code can be executed as soon as it's written. Because it doesn't need to take a long time to compile like Java, C++, or Rust, releasing code to production is typically faster.

Why not Python?

Python might not be the best choice for a project if:

- The code needs to run fast. Python code executes very slowly, which is why performance critical applications like PC games aren't written in Python.
- The codebase will become large and complex. Due to it's dynamic type system, Python code can be harder to keep clean of bugs.
- The application needs to be distributed directly to non-technical users. They would have to install Python in order to run your code, which would be a huge inconvenience.

Python 2 vs Python 3

The Python ecosystem suffers from split personality syndrome. Python 3 was released on December 3rd, 2008, but over a decade later the web is still full of Python 2 dependencies, scripts and tutorials.

In this course, we use Python 3 - just like any good citizen should these days.

One of the most obvious breaking changes between Python 2 and 3 is the syntax for printing text to the console.

Python 2

```
print "hello world"
```

Python 3

```
print("hello world")
```

Assignment

Update the code from Python 2 to Python 3 syntax.

Should you use Python 2 or 3?

As you've probably guessed, you should always use Python 3 going forward!

Python 2 and 3 are similar, but Python 3 contains some small but significant changes that are *not* backward compatible with the 2.x versions.

The Zen of Python

Tim Peters, a long time Pythonista describes the guiding priciples of Python in his famous short piece, The Zen of Python.

> *Beautiful is better than ugly.*
>
> *Explicit is better than implicit.*
>
> *Simple is better than complex.*
>
> *Complex is better than complicated.*
>
> *Flat is better than nested.*
>
> *Sparse is better than dense.*

> *Readability counts.*
>
> *Special cases aren't special enough to break the rules.*
>
> *Although practicality beats purity.*
>
> *Errors should never pass silently.*
>
> *Unless explicitly silenced.*
>
> *In the face of ambiguity, refuse the temptation to guess.*
>
> *There should be one-- and preferably only one --obvious way to do it.*
>
> *Although that way may not be obvious at first unless you're Dutch.*
>
> *Now is better than never.*
>
> *Although never is often better than* right *now.*
>
> *If the implementation is hard to explain, it's a bad idea.*
>
> *If the implementation is easy to explain, it may be a good idea.*
>
> *Namespaces are one honking great idea -- let's do more of those!*

Python Comments

Like in other languages, comments are non-executable text in your code, as a note or reminder to yourself and other developers.

Comments are ignored by the Python compiler when your code is compiled.

```
# this code prints a greeting
print('hello world') # you can also use comments inline
```

Python Multi-line Comments

Block (aka multi-line) comments don’t *technically* exist in Python. That said, there is a workaround.

```
""" the code found below
will print 'Hello, World!' to the console """
print('Hello, World!')
```

Wrapping a comment inside a set of triple quotes creates a "bare string literal", or a "docstring:. Technically, these strings are read by the interpreter, because a value is being created. However, it's completely safe because the rest of the code can't *access* the value. Triple quotes for block comments is fairly common practice in production code.

Variables

In this chapter, we'll go over the basic variable types in Python.

Let's build the back-end of a Movie Review app

For the next few chapters we'll create the building blocks of a back-end web server. The purpose of the server will be to supply up-to date information about the state of our Movie review mobile app. For example, the server will:

- Store and serve user information (emails, names, passwords)
- Provide authentication mechanisms
- Store and serve movie reviews

- Recommend movies based on choices
- Etc.

Python Strings

In Python, a string is written in double quotes. e.g. `"Hello"`, or alternately within single quotes like `'Hello'`.

The `len(s)` function returns the length of a string.

```
print(len("Hello"))
# prints "5"
```

Square brackets are used to access individual chars inside a string. The chars are numbered starting with 0, and running up to length-1.

```
greeting = 'Hello'
greeting[0]    ## 'H'
greeting[1]    ## 'e'
greeting[2]    ## 'l'
greeting[3]    ## 'l'
greeting[4]    ## 'o'
## you can also get the last char at length-1
greeting[len(greeting) - 1] ## 'o'
greeting[5]    ## ERROR, index out of bounds
```

You'll also notice that we don't use any keywords like `let`, `var`, or `const` to create new variables. We just use the assignment operator: =.

Negative String Index

As an alternative, Python supports using negative numbers to index into a string: `-1` points to the last character, -2 is the second to last,etc.

With that in mind, `s[-1]` is the same as `s[len(s)-1]`, the former just being easier to read.

Assignment

Print the 2nd to last character of one of our user's `usernames` using the negative number syntax we just discussed.

Python Numbers

In Python, numbers without a decimal part are called `Integers`. Contrast this to JavaScript where all numbers are just a `Number` type.

Integers are simply whole numbers, positive or negative. For example, 3 and `-3` are both examples of integers.

Arithmetic can be performed as you might expect:

Addition

```
2 + 1
# 3
```

Subtraction

```
2 - 1
# 1
```

Multiplication

```
2 * 2
# 4
```

Division

```
3 / 2
# 1.5 (a float)
```

This one is actually a bit different - division on two integers will actually produce a `float`. A float is, as you may have guessed, the number type that allows for decimal values.

Floor division

Python has great out-of-the-box support for mathematical operations. This, among other reasons, is why it has had such success in artificial intelligence, machine learning, and data science applications.

Floor division is like normal division except the result is floored afterwards, which means the remainder is removed. As you'd expect, this means the result is an `integer` instead of a `float`.

```
7 // 3
# 2 (an integer)
```

Exponents

Python has built-in support for exponents - something most languages require a `math` library for.

```
# reads as "three squared" or
# "three raised to the second power"
3 ** 2
# 9
```

Plus Equals

Python makes reassignment easy when doing math. In JavaScript or Go you might be familiar with the ++ syntax for incrementing a number variable. In Python, we use the += operator instead.

```
star_rating = 4
star_rating += 1
# star_rating is now 5
```

Assignment

A user added 5 new movies to our movie review app! On line 3 use the += operator to increment the `number_of_movies_in_app` by `5`.

Plus Equals

Python makes reassignment easy when doing math. In JavaScript or Go you might be familiar with the ++ syntax for incrementing a number variable. In Python, we use the += operator instead.

```
star_rating = 4
star_rating += 1
# star_rating is now 5
```

Assignment

A user added 5 new movies to our movie review app! On line 3 use the += operator to increment the `number_of_movies_in_app` by `5`.

```
number_of_movies_in_app = 156

# increment number_of_movies_in_app here
number_of_movies_in_app += 5
print(number_of_movies_in_app)
```

Scientific Notation

As we covered earlier, a `float` is a positive or negative number with a fractional part.

You can add the letter `e` or `E` followed by a positive or negative integer to specify that you're using scientific notation.

```
print(16e3)
# Prints 16000.0

print(7.1e-2)
# Prints 0.071
```

If you're not familiar with scientific notation, it's a way of expressing numbers that are too large or too small to conveniently write normally.

In a nutshell, the number following the e specifies how many places to move the decimal to the right for a positive number, or to the left for a negative number.

Logical Operators

You're probably familiar with the logical operators `AND` and `OR`. In JavaScript and Golang the logical `AND` operator is `&&` while the logical `OR` operator is `||`.

Logical operators deal with boolean values, `True` and `False`.

The logical `AND` operator requires that *both* inputs are `True` on order to return `True`. The logical `OR` operator only requires that *at least one* input is `True` in order to return `True`.

For example:

```
True AND True  = True
True AND False = False
False AND False = False

True OR True  = True
True OR False = True
False OR False = False
```

Python Syntax

Python is cool in that it doesn't use confusing symbols like `&&` and `||`, it literally uses the keywords `and` and `or`.

```
print(True and True)
# prints True

print(True or False)
# prints True
```

Nesting with parentheses

As you would expect, you can nest logical expressions just like any other code.

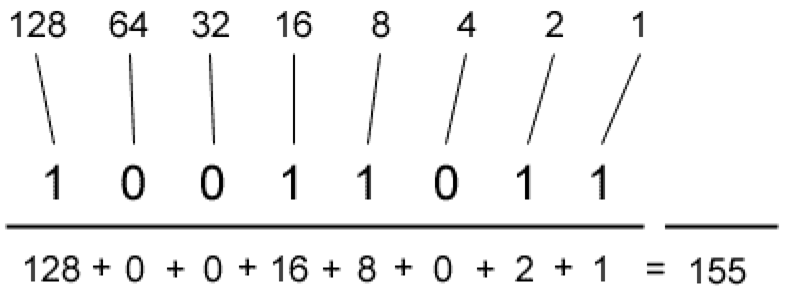
```
print((True or False) and False)
# prints False
```

Binary Numbers

Binary numbers are just "base 2" numbers. They work the same way as "normal" base 10 numbers, but with 2 symbols intead of 10.

Each `1` in a binary number represents a greater multiple of 2. In a 4-digit number, that means you have the eight's place, the four's place, the two's place, and the one's place. Similar to how in decimal you would have the thousandth's place, the hundreth's place, the ten's place, and the one's place.

- `0001` = 1
- `0010` = 2
- `0011` = 3
- `0100` = 4
- `0101` = 5
- `0110` = 6
- `0111` = 7
- `1000` = 8



Binary in Python

You can write an integer in Python using binary syntax using the `0b` prefix

```
print(0b0001)
# Prints 1

print(0b0101)
# Prints 5
```

Bitwise "&" Operator

Bitwise operators are similar to logical operators, but instead of operating on boolean values, they apply the same logic to all the bits in a value. For example, say you had the numbers 5 and 7 represented in binary. You could perform a bitwise AND operation that would result in 5

- `0101` is 5
- `0111` is 7

```
0101

AND

0111

=

0101
```

A `1` in binary is the same as `True`, while `0` is `False`. So really a bitwise operation is just a bunch of logical operations that are completed in tandem.

`&` is the bitwise AND operator in Python. So `5 & 7 = 5`, while `5 & 2 = 0`

```
0101 = 5
0010 = 2
&
0000 = 0
```

Movie Review Permissions

It's common practice to store user permissions as binary values. Think about it, if I have 4 different permissions a user can have, then I can store that as a 4-digit binary number, and if a certain bit is present, I know the permission is enabled.

Let's pretend we have 4 permissions, `can_create_movie`, `can_review_movie`, `can_delete_movie`, and `can_edit_movie` represented by `0b0000`. For example, if a user only has the `can_create_movie` permission, their binary permissions would be `0b1000`. A user with `can_review_movie` and `can_edit_movie` would be `0b0101`.

In order to check for, say, the `can_review_movie` permission, we can perform a bitwise `AND` operation on the user's permissions and the enabled `can_review_movie` bit (`0b1000`). If the result is `0b1000` again, we know they have that specific permission!

Assignment

Assign a binary value to the `user_permissions` variable so that the user will have the `can_review_movie` permission and the `can_delete_movie` permission.

```
user_permissions = 0b0110

# DON'T EDIT BELOW THIS LINE

can_create_movie = 0b1000
can_review_movie = 0b0100
can_delete_movie = 0b0010
can_edit_movie = 0b0001

user_can_create_movie = user_permissions & can_create_movie ==
can_create_movie
user_can_review_movie = user_permissions & can_review_movie ==
can_review_movie
user_can_delete_movie = user_permissions & can_delete_movie ==
can_delete_movie
user_can_edit_movie = user_permissions & can_edit_movie ==
can_edit_movie

print("user_can_create_movie: {}".format(user_can_create_movie))
print("user_can_review_movie: {}".format(user_can_review_movie))
print("user_can_delete_movie: {}".format(user_can_delete_movie))
print("user_can_edit_movie: {}".format(user_can_edit_movie))
```

Not

We skipped a very important logical operator - `not`. The `not` operator reverses the result. It returns `False` if the input was `True` and vice-versa.

```
print(not True)
# Prints: False

print(not False)
# Prints: True
```

Lists

Python's lists are the equivalent of JavaScript's arrays or Golang's slices. Lists are written as a list of comma-separated values (items) between square brackets.

One important thing about a list is that items in a list are *not* required to be of the same type.

```
sciency_stuff = ['physics', 'chemistry', 69, 42.0]
numbers = [1, 2, 3, 4, 5]
```

Index into a list

Just like we already saw with strings, you can get items from a list using square brackets.

```
sciency_stuff = ['physics', 'chemistry', 69, 42.0]
print(sciency_stuff[0])
# Prints: 'physics'

print(sciency_stuff[-1])
# Prints: 42.0
```

Append to a list

```
drinks = []
drinks.append('lemonade')
print(drinks)
# Prints: ['lemonade']
drinks.append('root beer')
print(drinks)
# Prints: ['lemonade', 'root beer']
```

Assignment

1. Create an empty list called 'movies'
2. Print the list
3. Append 'the dark knight` to your list of movies

4. Print the list
5. Append 'the notebook' to your list of movies
6. Print the list
7. Print the item at index `0` of the movies list

```
movies = []
print(movies)
movies.append('the dark knight')
print(movies)
movies.append('the notebook')
print(movies)
print(movies[0])
```

Lists Length

The `len` function returns the current length of a list.

```
foods = ['burger', 'fries', 'pizza']
print(len(foods))
# Prints: 3
```

Assignment

We want to find a user on our movie review app so we can ask them for feedback. Our first users (at the beginning of the usernames list) aren't familiar with our newest features, and our newest users haven't been using the app long enough. We need someone from the middle of the list!

1. Create a variable called `num_users` and set it equal to the length of the `usernames` list.
2. Print the `num_users` variable
3. Use floor division (`//`) to get the index of the username halfway through the list rounded down.
4. Print that username

```
usernames = [
    "johndoe123",
    "billyrae456",
    "sallysue124",
    "pratham",
    "preston",
    "wagslane",
    "jimmyjohn",
    "bopeep",
    "strightkilla",
    "reddyman",
    "singsonger",
    "killingeet",
    "movieluvver",
    "filmenthus",
    "yoyoyoyo123",
]
num_users = len(usernames)
print(num_users)
req_user = num_users//2
print(usernames[req_user])
```

Looping over a list

Lists aren't super useful if we can't iterate over their items! Python has, in my opinion, the most elegant syntax for list iteration.

```
names = ["john", "james", "sal", "suzie"]
for name in names:
    print(name)
# Prints:
# john
# james
# sal
# suzie
```

The `for` and `in` keywords let us directly loop over all the *values* in a list. Note that *whitespace matters*. The body of the loop must be indented, otherwise you'll get a syntax error.

Assignment

We want to create some customized greeting emails for the users of our app! Loop over the list of provided names and print `"Hello NAME! Welcome to movie reviews plus!"` where `NAME` is the name in the list.

Hint

Python has a format method that makes it easy to use variables in a larger string. You call the `.format()` method at the end of a string, and it will replace any empty squiggly brackets `{}` with the parameters provided.

```
print("My first name is: {}. My last name is: {}".format("lane",
"wagner"))
# Prints: "My first name is: lane. My last name is: wagner"
```

```
names = [
    "Johnny",
    "Billy",
    "Pratham",
    "Sam",
    "Jake",
    "Shana",
    "Camila",
    "Nicole",
    "Lafawnda",
]
for name in names:
    print("Hello {}! Welcome to movie reviews plus!".format(name))
```

```
OUTPUT

Hello Johnny! Welcome to movie reviews plus!

Hello Billy! Welcome to movie reviews plus!

Hello Pratham! Welcome to movie reviews plus!

Hello Sam! Welcome to movie reviews plus!

Hello Jake! Welcome to movie reviews plus!

Hello Shana! Welcome to movie reviews plus!

Hello Camila! Welcome to movie reviews plus!

Hello Nicole! Welcome to movie reviews plus!

Hello Lafawnda! Welcome to movie reviews plus!
```

List Operations - Concatenate

Concatenating two lists (smushing them together) is really easy in Python, just use the + operator.

```
all = [1, 2, 3] + [4, 5, 6]
print(all)
# Prints: [1, 2, 3, 4, 5, 6]
```

Assignment

Our movie review app allows users to keep lists of their favorite movies. We need to add them all to one giant list so we can keep track of the totals.

1. Create a new list that has John's movies, followed by Jack's, followed by Breanna's.
2. Print the total number of movies using the `len` function
3. Print the list containing all the movies

```
johns_favorites = ["inception", "spiderman"]
jacks_favorites = ["that thing you do", "midsommar"]
breannas_favorites = ["seven pounds", "how to lose a guy in 10 days"]
toal_favorites = johns_favorites + jacks_favorites + breannas_favorites
print(len(toal_favorites))
print(toal_favorites)
```

```
OUTPUT

6

['inception', 'spiderman', 'that thing you do', 'midsommar', 'seven
pounds', 'how to lose a guy in 10 days']
```

List Operations - Contains

Checking whether a value exists in a list is also really easy in Python, just use the `in` keyword.

```
fruits = ["apple", "orange", "banana"]
print("banana" in fruits)
# Prints: True
```

Assignment

Our users have requested a feature in our app that will allow them to type in a movie and check if it's in our list of top movies for the week.

Take a look at the 3 print statements. Replace the static boolean values in the `format` parameters with a statement that checks if the movie in the text is in the `top_movies` list.

```
top_movies = [
    "inception",
    "batman begins",
    "oh brother where art thou",
    "tangled",
    "happy feet",
    "ice age",
    "toy story",
]

print("inception is a top movie: {}".format("inception" in top_movies))
print("tangled is a top movie: {}".format("tangled" in top_movies))
print("the dark knight is a top movie: {}".format("the dark knight" in
top_movies))
```

```
OUTPUT
inception is a top movie: True
tangled is a top movie: True
the dark knight is a top movie: False
```

List Loops - Indexes

While most of the time when you're working with lists, you'll want to iterate over the items, sometimes you'll want the indexes themselves.

In Python you can iterate over a range of numbers in the following way.

```
for i in range(0, 3):
    print(i)
# Prints:
# 0
# 1
# 2
```

Assignment

We need to show our users what the top five movies are, but not just *which* movies make the cut, our users want to know the ranking.

The `top_five` list has the top five movies in rank order from 1-5. Loop over the list and print the following for each entry:

```
MOVIE is rank RANK on movie reviews plus!
```

Where `MOVIE` is the name of the movie and `RANK` is the index in the list *plus 1*. Remember, the indexes in the list are 0-4 but we want them to display as 1-5.

```
top_five = [
    "avatar",
    "the lion king",
    "dead poet's society",
    "christmas vacation",
    "elf",
]
for i in range(0, len(top_five)):
   print("{} is rank {} on movie reviews plus!".format(top_five[i],
i+1))
```

```
OUTPUT
avatar is rank 1 on movie reviews plus!
the lion king is rank 2 on movie reviews plus!
dead poet's society is rank 3 on movie reviews plus!
christmas vacation is rank 4 on movie reviews plus!
elf is rank 5 on movie reviews plus!
```

Append

The `append` method is the equivalent of JavaScript's `.push` or Golang's `append` function.

It takes a single argument, and adds that value to the end of the list it's called on.

```
veggies = []
veggies.append('carrot')
# ['carrot']
veggies.append('cucumber')
# ['carrot', 'cucumber']
```

Assignment

We have a numeric list of our customer's star ratings of various movies. We need to create a `labels` list that has a more presentable version. The integer 5 becomes the string `"5 stars"`, 4 becomes `"4 stars"`, etc.

1. Create an empty `labels` list
2. Loop over each star rating
3. Append the corresponding string for each star label. Use the `.format` method.
4. Print the `labels` list

```
star_ratings = [5, 3, 2, 2, 5, 4, 1, 2, 5]
labels = []
for i in star_ratings:
    labels.append("{} stars".format(i))

print(labels)
```

```
OUTPUT
['5 stars', '3 stars', '2 stars', '2 stars', '5 stars', '4 stars', '1
stars', '2 stars', '5 stars']
```

Slicing lists

Python makes it easy to slice and dice lists to work only with the section you care about. One way to do this is to use the simple slicing operator, which is just a colon `:`.

With this operator, you can specify where to start and end the slice, and how to step through the original. List slicing returns a *new list* from the existing list.

The syntax is as follows:

```
Lst[ Initial : End : IndexJump ]
```

```
scores = [50, 70, 30, 20, 90, 10, 50]
# Display list
print(scores[1:5:2])
# Prints [70, 20]
```

The above reads as "give me a slice of the `scores` list from index 1, up to but not including 5, skipping every 2nd value. *All of the sections are optional*.

```
scores = [50, 70, 30, 20, 90, 10, 50]
# Display list
print(scores[1:5])
# Prints [70, 30, 20, 90]
```

```
scores = [50, 70, 30, 20, 90, 10, 50]
# Display list
print(scores[1:])
# Prints [70, 30, 20, 90, 10, 50]
```

Assignment

1. First, print a slice of the `movies` list that starts with the third item in the list.
2. Next, print a slice of the `movies` list that ends with the third item from the end of the list.
3. Last, print a slice of the `movies` list that skips every odd numbered index.

```
movies = [
    "oh brother where art thou",
    "oceans eleven",
    "fight club",
    "the island",
    "shutter island",
    "the magnificent seven",
]
print(movies[2:])
print(movies[:-2])
print(movies[::2])
```

```
OUTPUT

['fight club', 'the island', 'shutter island', 'the magnificent seven']
['oh brother where art thou', 'oceans eleven', 'fight club', 'the
island']
['oh brother where art thou', 'fight club', 'shutter island']
```

List deletion

Python has a built-in keyword `del` that deletes items from objects. In the case of a list, you can delete specific indexes or entire slices.

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# delete the fourth item
del nums[3]
print(nums)
# Output: [1, 2, 3, 5, 6, 7, 8, 9]

# delete items from 2nd to 3rd
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del nums[1:3]
print(nums)
# Output: [1, 4, 5, 6, 7, 8, 9]

# delete all elements
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del nums[:]
print(nums)
# Output: []
```

Assignment

1. Delete the first movie from the list
2. Print the list of movies
3. Delete the last two movies from the list as a slice
4. Print the list of movies

```
movies = [
    "oh brother where art thou",
    "oceans eleven",
    "fight club",
    "the island",
    "shutter island",
    "the magnificent seven",
]
del movies[0]
print(movies)
del movies[-2:]
print(movies)
```

```
OUTPUT
['oceans eleven', 'fight club', 'the island', 'shutter island', 'the
magnificent seven']
['oceans eleven', 'fight club', 'the island']
```

If, Elif, Else

Conditional logic in Python is similar to other languages with a couple quirks. The only unique thing is that rather than an `else if` statement, Python has an `elif` keyword that accomplishes the same thing.

```
if height > 100:
   print("you're tall!")
elif height > 60:
    print("you're average height")
elif height > 50:
    print("you could be taller")
else:
   print("you're short")
```

Assignment

We need to keep mature movies away from children!

Iterate over all the ages in the list, if the age is greater than or equal to `18`, print `You're AGE. You can see mature films.` where `AGE` is the actual age.

Hint

Inequality in Python works similar to other C-like languages.

```
10 > 5 # True - greater than
5 >= 5 # True - greater than or equal to
10 == 5 # False - Equal to
10 != 5 # True  - Not equal to
```

```
ages = [10, 18, 24, 6, 70, 65, 45, 23, 22, 10, 18, 19]
for age in ages:
    if age >= 18:
        print ("You're {}. You can see mature films.".format(age))
    #else:
        #print ("You're too young to watch this")
```

```
OUTPUT
You're 18. You can see mature films.
You're 24. You can see mature films.
You're 70. You can see mature films.
You're 65. You can see mature films.
You're 45. You can see mature films.
You're 23. You can see mature films.
You're 22. You can see mature films.
You're 18. You can see mature films.
You're 19. You can see mature films.
```

Assignment

Let's do the same thing, but this time we'll break down our messaging a bit more.

1. Iterate over all the ages again.
2. If an age is greater than or equal to `18`, print `You're AGE. You can see mature films.` where `AGE` is the actual age.
3. Otherwise, if an age is greater than or equal to `16`, print `You're AGE. You can see teen films.`
4. Otherwise print `You're AGE. You can see kid films.`

```
ages = [10, 18, 24, 6, 3, 16, 70, 65, 45, 23, 22, 10, 18, 19, 16, 17,
4, 2]
for age in ages:
    if age >= 18:
        print ("You're {}. You can see mature films.".format(age))
    elif age >= 16:
        print ("You're {}.You can see teen films.".format(age))
    else:
        print ("You're {}.You can see kid films.".format(age))
```

```
OUTPUT
You're 10.You can see kid films.
You're 18. You can see mature films.
You're 24. You can see mature films.
You're 6.You can see kid films.
You're 3.You can see kid films.
You're 16.You can see teen films.
You're 70. You can see mature films.
You're 65. You can see mature films.
You're 45. You can see mature films.
You're 23. You can see mature films.
You're 22. You can see mature films.
You're 10.You can see kid films.
You're 18. You can see mature films.
You're 19. You can see mature films.
You're 16.You can see teen films.
You're 17.You can see teen films.
You're 4.You can see kid films.
You're 2.You can see kid films.
```

List of lists

Nested lists are matrices, you can think of them like a grid.

```
chess_board = [
    ['Rook', 'Knight', 'Bishop', 'King', 'Queen', 'Bishop', 'Knight',
'Rook'],
    ['Pawn','Pawn','Pawn','Pawn','Pawn','Pawn','Pawn','Pawn'],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    ['Pawn','Pawn','Pawn','Pawn','Pawn','Pawn','Pawn','Pawn'],
    ['Rook', 'Knight', 'Bishop', 'King', 'Queen', 'Bishop', 'Knight',
'Rook'],
]

for row in chess_board:
    for space in row:
        print(space)
        # Prints:
        # Rook
        # Knight
        # ...
```

In Python, `None` is similar to JavaScript's `null` or Golang's `Nil`. It's a falsy object value that represents the idea that "nothing's here".

Assignment

We have a nested list of movie ratings in our database. Each list in the matrix is the ratings of the movies in the playlist.

Loop over each playlist, and for each playlist you will either be printing `mature playlist` if it contains an "r" rating, otherwise you will print `safe for kids`. You are printing one statement *per playlist, not per movie*.

```
users_playlist_ratings = [
    ["pg", "r", "pg-13", "r"],
    ["g", "g", "g"],
    ["pg", "g", "pg"],
    ["pg-13", "pg-13", "pg"],
    ["pg-13", "pg-13", "r", "r"],
]
for playlist in users_playlist_ratings:
    if "r" in playlist:
        print("mature playlist")
    else:
        print("safe for kids")
```

```
OUTPUT
mature playlist
safe for kids
safe for kids
safe for kids
mature playlist
```

```
OTHER SOLUTION
users_playlist_ratings = [
    ["pg", "r", "pg-13", "r"],
    ["g", "g", "g"],
    ["pg", "g", "pg"],
    ["pg-13", "pg-13", "pg"],
    ["pg-13", "pg-13", "r", "r"],
]

for playlist in users_playlist_ratings:
    mature = False
    for rating in playlist:
        if rating == "r":
            mature = True
    if mature:
        print("mature playlist")
    else:
        print("safe for kids")
```

NoneType

Like we mentioned in the last exercises, the `None` keyword is used to define a null variable or object. In Python, the `None` keyword is an object, and it is a data type of the class `NoneType`.

Random Fact

All variables that are assigned to `None` point to the *same* instance of a `NoneType` object. New instances of `None` are never created.

Functions

As you might have guessed, Python supports the ability to define your own functions. You'll use the `def` keyword. Similar to other blocks like `if` and `else`, the body of the function comes after a colon and is indented once.

```
def main():
    print('hello world')

main()
```

Assignment

We've broken out our logic that takes the number of stars and returns a user-friendly label. We need to call the function!

At the end of the existing code, loop over all the star counts in the `stars` list. For each one, print the result of the `get_label` function called with that rating.

```
stars = [3, 2, 1, 5, 4, 10]


def get_label(num_stars):
    if num_stars == 5:
        return "5 stars!"
    elif num_stars == 4:
        return "4 stars!"
    elif num_stars == 3:
        return "3 stars!"
    elif num_stars == 2:
        return "2 stars!"
    elif num_stars == 1:
        return "1 stars!"
    else:
        return "error"
for star in stars:
    print (get_label(star))
```

```
OUTPUT
3 stars!
2 stars!
1 stars!
5 stars!
4 stars!
error
```

Order of functions

Python code is interpreted line-by-line. That means that if you define a function, you can't call that function until after the definition.

Assignment

The `main()` function is a convention in many programming languages to specify the entrypoint of an application.

Ours isn't working! Fix the bug.

```
main()
def main():
    print("starting the movie review server")
output:
error
========================================================
After FIx
========================================================
def main():
    print("starting the movie review server")
main()
output:
starting the movie review server
```

Order of functions

All functions *must* be defined before they're used.

You might think this would make structuring Python code difficult because the order in which the functions are declared can quickly become so dependent on each other that writing anything becomes impossible.

As it turns out, most Python developers solve this problem by simply defining all the functions first, then finally calling the entrypoint function *last*. If you do that, then the order of all the functions are declared in *doesn't matter*.

```
def func1():
    func2()

def func2():
    func3()

def func3():
    print("I'm function 3")

func1() # entrypoint
```

Parameters vs arguments

Parameters are the names used when *defining* a function. Arguments are the things which are supplied to any function or method call, while the function or method code refers to the arguments by their parameter names.

Arguments are the actual values that go into the function, say `42.0`, `"the dark knight"`, or `True`. Parameters are the names we use in the function definition to refer to those values, which at the time of writing the function, could be anything.

That said, it is important to understand that this is all semantics, and frankly developers are really lazy with these definitions. You'll often hear the words arguments and parameters used interchangeably.

Multiple return values

In Python, we can return multiple values from a function. All we need to do is separate each value by a comma.

```
# returns email, age, and status of the user
def get_user():
    return "name@domain.com", 21, "active"

email, age, status = get_user()
print(email)
# Prints: "name@domain.com"
print(age)
# Prints: 21
print(status)
# Prints: "active"
```

```
def get_user():
    return "name@domain.com", 21, "active"

# this works, and by convention you should NOT use the underscore
variable later
email, _, _ = get_user()
print(email)
# Prints: "name@domain.com"
print(_)
# Prints: "active"
```

Assignment

We need to take profanity out of some of our movie titles for youngsters on the platform! Complete the `filter_movies` function. It takes a list of movie titles as input and returns 2 new lists. The first list it returns contains the same title but with all instances of the words `shoot` and `dang` removed. The second list returned is the number of `shoot` and `dang` words that were removed.

1. Create the empty lists that you'll return at the end of the function
2. For each movie title:
    a. Split the movie title on whitespace using the `split()` method to get a list of words in the title (see below for help)
    b. Create a new list that will contain all the non-bad words for this movie
    c. Create a counter variable and set it to `0`. We'll increment this when we remove words from this title
    d. For each word in the title:
        i. If the word is `dang` or `shoot` increment the counter
        ii. If it isn't a bad word, instead add the word to the non-bad word list you created
    e. Join the list of non-bad words into a single string (see below for help)
    f. Append the new clean title to the final list of filtered movies titles
    g. Append the counter of bad words removed to its list
    h. Return the titles first, then the counters

Hints

Split string

The `split()` method is called on a string. If you pass it no arguments, it will just split the words in the string on the whitespace.

```
words = "hello there sam".split()
print(words)
# Prints: ["hello", "there", "sam"]
```

Join strings

The `join()` method is called on a delimiter (what goes between all the words in the list), and takes a list of strings as input.

```
sentence = " ".join(["hello", "there", "sam"])
print(sentence)
# Prints: "hello there sam"
```

```
def filter_movies(movies):
    filtered_movies = []
    words_removed = []
    for movie in movies:
        words = movie.split()
        new_words = []
        removed = 0
        for word in words:
            if word == "dang" or word == "shoot":
                removed += 1
            else:
                new_words.append(word)
        filtered_movies.append(" ".join(new_words))
        words_removed.append(removed)

    return filtered_movies, words_removed


# Don't edit below this line


def main():
    movies = [
        "avril lavigne: the best dang tour - live in toronto",
        "love and pain and the whole dang thing",
        "kiss me, dang it",
        "frozen",
        "donkey kong",
        "the shoot and the dang",
        "ghosts with shoot jobs",
    ]
    filtered_movies, words_removed = filter_movies(movies)
    print(filtered_movies)
    print(words_removed)


main()
```

```
OUTPUT
['avril lavigne: the best tour - live in toronto', 'love and pain and
the whole thing', 'kiss me, it', 'frozen', 'donkey kong', 'the and
the', 'ghosts with jobs']
[1, 1, 1, 0, 0, 2, 1]
```

Ignoring a return value

You need to accept all the return values of a function, if you don't you'll get a syntax error from the interpreter. That said, there's a convention where you can use the underscore _ character as a variable name that's meant to be ignored. It's just a convention though, you could actually use that underscore later in the code if you were sloppy.

```
def get_user():
    return "name@domain.com", 21, "active"

# this results in an error
email, age = get_user()
```

Recursion

Python supports recursion, that is, functions that call themselves.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
```

Assignment

Let's write a recursive implementation of the factorial operation. A factorial is a mathematical operation denoted by an exclamation point: `!`. We'll use this function to determine the total number of possible ways a list of movies could be ranked in order. It just so happens that the total number of orderings is `n!` (n-factorial) where `n` is the number of movies.

```
2! = 1 * 2             = 2
3! = 1 * 2 * 3         = 6
4! = 1 * 2 * 3 * 4     = 24
5! = 1 * 2 * 3 * 4 * 5 = 120
```

Steps

Complete the `factorial()` function.

1. If `x` is `0` just return `1`
2. Otherwise, return the result of `x` multiplied by the factorial of `x - 1`

```
def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(x-1)



# Don't edit below this line


def main():
    print("{} orderings for {} movies".format(factorial(5), 5))
    print("{} orderings for {} movies".format(factorial(6), 6))
    print("{} orderings for {} movies".format(factorial(10), 10))
    print("{} orderings for {} movies".format(factorial(12), 12))
    print("{} orderings for {} movies".format(factorial(15), 15))


main()
```

```
OUTPUT
120 orderings for 5 movies
720 orderings for 6 movies
3628800 orderings for 10 movies
479001600 orderings for 12 movies
1307674368000 orderings for 15 movies
```

List Practice - Lowercase

Python has `lower()` and `upper()` methods that can be called on a string.

```
print("Hey there!".lower())
# Prints: "hey there!"

print("Hey there!".upper())
# Prints: "HEY THERE!"
```

Assignment

Complete the `lower_list()` function. It should take a list of strings and return a new list with the same strings but all lowercased.

In our case we'll be lowercasing movies titles so we can have consistency for our machine learning inputs - it only works on lowercase characters.

```
def lower_list(strings):
    new_strings = []
    for string in strings:
        new_strings.append(string.lower())
    return new_strings


# Don't edit below this line


def main():
    print(
        lower_list(
            ["Oh brother where art thou", "Hello Dolly", "Monsters
INC", "FARGO"]
        )
    )


main()
```

```
OUTPUT
['oh brother where art thou', 'hello dolly', 'monsters inc', 'fargo']
```

List Practice - Is Number

In Python you can compare strings using the inequality operators, e.g. <, >, <=, and >=. This lets you check if a character is within a specific range.

For example:

```
"0" < "1" # True
"0" < "9" # True
"0" > "9" # False
"1" >= "0" # True
```

Python string comparison is performed using the *characters* in both strings. The characters in both strings are compared one by one. When different characters are found, their Unicode value is compared. The character with the lower Unicode value (numeric representation) is considered smaller.

Assignment

Our users want to know how many movies have numbers in their titles for... research purposes? I guess?

Anyhow, complete the `contains_num` function. It takes a title as input. Loop over each character in the title, and if you find a number return `True` . If no numbers are found return `False`.

Hint

```
for char in string:
```

```
def contains_num(title):
    for char in title:
        if char <= "9":
            return True
    return False


# Don't edit below this line


def main():
    print(contains_num("farenheit 451"))
    print(contains_num("batman"))
    print(contains_num("44 year old virgin"))
    print(contains_num("spiderman"))
    print(contains_num("oceans 11"))
    print(contains_num("50 first dates"))


main()
```

```
OUTPUT:
True
False
True
False
True
True
```

None Return

When no return value is specified in a function, it will return `None`. The following code snippets all return exactly the same thing:

```
def my_func():
    print("I do nothing")
    return None
```

```
def my_func():
    print("I do nothing")
    return
```

```
def my_func():
    print("I do nothing")
```

Dictionaries

Dictionaries in Python are used to store data values in `key->value` pairs. They're similar to JavaScript's object literals or Golang's maps.

```
car = {
  "brand": "Tesla",
  "model": "3",
  "year": 2019
}
```

Assignment

Complete the `get_movie_record` function. It takes a movie's `title`, the number of `stars`, and the `username` of the person who gave the star rating. It should return a dictionary with 4 fields.

- title
- stars
- username
- id

Where the `id` are the `title` and the `username` concatenated together for uniqueness. For example, `toy story` and `hanks123` would make the id `toy storyhanks123`.

```python
def get_movie_record(title, stars, username):
    return {
        "title": title,
        "stars": stars,
        "username": username,
        "id": title + username
    }


# Don't edit below this line
def main():
    rec = get_movie_record("oh brother where art thou", 3, "wagslane")
    print_rec(rec)

    rec = get_movie_record("frozen", 5.5, "elonmusk")
    print_rec(rec)

    rec = get_movie_record("toy story", 4, "prince")
    print_rec(rec)


def print_rec(rec):
    print("title: {}".format(rec["title"]))
    print("stars: {}".format(rec["stars"]))
    print("username: {}".format(rec["username"]))
    print("id: {}".format(rec["id"]))
    print("---")


main()
```

```
OUTPUT

title: oh brother where art thou

stars: 3

username: wagslane

id: oh brother where art thouwagslane

---

title: frozen

stars: 5.5

username: elonmusk

id: frozenelonmusk

---

title: toy story

stars: 4

username: prince

id: toy storyprince

---
```

Ordered or Unordered?

As of Python version `3.7`, dictionaries are ordered. In Python `3.6` and earlier, dictionaries were unordered.

Because dictionaries are ordered, the items have a defined order, and that order will *not* change.

Unordered means that the items didn't have a defined order, so you couldn't refer to an item by using an index.

Duplicate Keys

Because dictionaries rely on unique keys, you can't have two keys that are the same. If you try to use the same key twice, the value of the first will simply be overwritten.

Accessing Dictionary Values

Dictionary elements must be accessible somehow in code, otherwise they wouldn't be very useful.

A value is retrieved from a dictionary by specifying its corresponding key in square brackets. The syntax looks similar to indexing into a list.

```
car = {
    'make': 'tesla',
    'model': '3'
}
print(car['make'])
# Prints: tesla
```

Setting Dictionary Values

You don't need to create a dictionary with values already inside. It is common to create a blank dictionary then populate it later in the code. The syntax is the same, just pass a new key into the square brackets and use the assignment operator (=) to give that key a value.

```
names = ["jack bronson", "jill mcarty", "john denver"]

names_dict = {}
for name in names:
    names_arr = name.split()
    names_dict[names_arr[0]] = names_arr[1]

print(names_dict)
# Prints: {'jack': 'bronson', 'jill': 'mcarty', 'john': 'denver'}
```

Updating Dictionary Values

If you try to set a value to a key that already exists, you'll end up just updating the value.

```
names = ["jack bronson", "jack mcarty", "jack denver"]

names_dict = {}
for name in names:
    names_arr = name.split()
    names_dict[names_arr[0]] = names_arr[1]

print(names_dict)
# Prints: {'jack': 'denver'}
```

Deleting Dictionary Values

You can delete existing keys using the `del` keyword.

```
names_dict = {
    'jack': 'bronson',
    'jill': 'mcarty',
    'joe': 'denver'
}

del names_dict['joe']

print(names_dict)
# Prints: {'jack': 'bronson', 'jill': 'mcarty'}
```

Deleting keys that don't exist

Notice that if you try to delete a key that doesn't exist, you'll get an *error*.

```
names_dict = {
    'jack': 'bronson',
    'jill': 'mcarty',
    'joe': 'denver'
}

del names_dict['unknown']
# ERROR HERE, key doesn't exist
```
```

### Checking for existence

If you're unsure whether or not a key exists in a dictionary, use the `in` keyword.

```
cars = {
    'ford': 'f150',
    'tesla': '3'
}

print('ford' in cars)
# Prints: True

print('gmc' in cars)
# Prints: False
```

Assignment
```

All of our users review different movies. We want to count up all the movie titles and see how often different movies were reviewed. Complete the `count_titles` function. It takes a list of movie titles as input. It should return a dictionary where the keys are all the movie titles from the list, and the values are the counts of how many time each movie title appeared in the list.

```
def count_titles(movie_titles):
    titles_map = {}
    for movie_title in movie_titles:
        if movie_title in titles_map:
            titles_map[movie_title] += 1
        else:
            titles_map[movie_title] = 1
    return titles_map


# Don't edit below this line


def main():
    print(
        count_titles(
            [
                "frozen",
                "monsters inc",
                "frozen",
                "monsters inc",
                "cars",
                "monsters inc",
                "cars",
                "cars",
                "frozen",
                "frozen",
                "toy story",
                "frozen",
                "frozen",
            ]
        )
    )


main()
```

```
OUTPUT
{'frozen': 6, 'monsters inc': 3, 'cars': 3, 'toy story': 1}
```

Understanding:

1. So `movie_title in titles_map` will be `True` if the `movie_title` in that iteration of the loop already exists in the `titles_map` dictionary
2. In other words, the FIRST time you loop over a specific title, it will NOT be in the map
3. But the second time you hit that same title, it will already be there
4. So, if it's already there, you should add one to the count. If it's not there yet, you need to create it and set it to 1

Sets

Sets are like Lists, but they are unordered and they guarantee uniqueness. There can be no two of the same value in a set.

```
fruits = {'apple', 'banana', 'grape'}
print(type(fruits))
# Prints: <class 'set'>

print(fruits)
# Prints: {'banana', 'grape', 'apple'}
```

Adding values

```
fruits = {'apple', 'banana', 'grape'}
fruits.add('pear')
print(fruits)
# Prints: {'banana', 'grape', 'pear', 'apple'}
```

Empty set

Because the `{}` syntax creates an empty dictionary, to create an empty set, just use the `set()` function.

```
fruits = set()
fruits.add('pear')
print(fruits)
# Prints: {'pear'}
```

Iterate over values in a set (order is not guaranteed)

```
fruits = {'apple', 'banana', 'grape'}
for fruit in fruits:
    print(fruit)
    # Prints:
    # banana
    # grape
    # apple
```

Assignment

Complete the `remove_duplicates` function. It should take a list of movie titles and return a new `List` where there is at most one of each title. You can accomplish this by creating a set, adding all the movies to it, then iterating over the set and adding all the movies back to a List and returning the list.

```
def remove_duplicates(movieTitles):
    movieTitles_set = set()
    original = []
    for movietitle in movieTitles:
        movieTitles_set.add(movietitle)
    for originals in movieTitles_set:
        original.append(originals)
    return original


# Don't edit below this line


def main():
    final = remove_duplicates(
        [
            "frozen",
            "monsters inc",
            "frozen",
            "monsters inc",
            "cars",
            "monsters inc",
            "cars",
            "cars",
            "frozen",
            "frozen",
            "toy story",
            "frozen",
            "frozen",
        ]
    )
    final.sort()
    print(final)


main()
```

```
OUTPUT
['cars', 'frozen', 'monsters inc', 'toy story']
```

Errors and exceptions in Python

You've probably encountered some errors in your code from time to time if you've gotten this far in the course. In Python, there are two main kinds of distinguishable errors.

- syntax errors
- exceptions

Syntax errors

You probably know what there are by now. A syntax error is just the Python interpreter telling you that your code isn't adhering to proper Python syntax.

```
this will error
```

If I try to run that sentence as if it were valid code I'll get a syntax error:

```
this will error
      ^
SyntaxError: invalid syntax
```

Exceptions

Even if your code has the right syntax however, it may still cause an error when an attempt is made to execute it. Errors detected during execution are called "exceptions" and can be handled gracefully by your code. You can even raise your own exceptions when bad things happen in your code.

Python uses a try-except pattern for handling errors.

```
try:
  10 / 0
except Exception as e:
  print("exception excepted: {}".format(e))

# prints "division by zero"
```

The `try` block is executed until an exception is raised or it completes, whichever happens first. In this case, a "divide by zero" error is raised because division by zero is impossible. The `except` block is only executed if an exception is raised in the `try` block. It then exposes the exception as data (e in our case) so that the program can handle the exception gracefully without crashing.

Assignment

One of the calls to `get_movie_record` is throwing a `movie id not found` exception. Change the code to safely make each call within a try-except block. If an exception is raised, just print it.

```
def main():
    try:
       print(get_movie_record(1))
       print(get_movie_record(2))
```

```
        print(get_movie_record(3))
        print(get_movie_record(4))
    except Exception as e:
        print(e)


# Don't edit below this line


def get_movie_record(movie_id):
    if movie_id == 1:
        return {"name": "Apollo 13", "duration": 128}
    if movie_id == 2:
        return {"name": "2001: A Space Odyssey", "duration": 300}
    if movie_id == 3:
        return {"name": "Interstellar", "duration": 4000}
    raise Exception("movie id not found")
```

```
OUTPUT
{'name': 'Apollo 13', 'duration': 128}
{'name': '2001: A Space Odyssey', 'duration': 300}
{'name': 'Interstellar', 'duration': 4000}
movie id not found
```

Raising your own exceptions

Errors are *not* something to be scared of. Every program that runs in production deals with errors on a constant basis. Our job as developers is to handle the errors gracefully and in a way that aligns with our user's expectations.

When something in our own code happens that we don't expect, we should raise our own exceptions. For example, if someone passes some bad inputs to a function we write, we shouldn't be afraid to raise an exception to let them know they did something wrong.

Syntax

```
raise Exception("something bad happened")
```

Assignment

If a `movie_id` is passed into the `get_movie_record` function, we need to raise our own error to alert the caller that the movie they are looking for doesn't exist. The exception should say `movie id not found`.

```
def get_movie_record(movie_id):
    if movie_id == 1:
        return {"name": "Apollo 13", "duration": 128}
    if movie_id == 2:
        return {"name": "2001: A Space Odyssey", "duration": 300}
    if movie_id == 3:
        return {"name": "Interstellar", "duration": 4000}
    if movie_id >= 4:
        return ("movie id not found")


# Don't edit below this line


def main():
    try:
        print(get_movie_record(1))
        print(get_movie_record(2))
        print(get_movie_record(3))
        print(get_movie_record(4))
    except Exception as e:
        print(e)


main()
```

```
OUTPUT
{'name': 'Apollo 13', 'duration': 128}
{'name': '2001: A Space Odyssey', 'duration': 300}
{'name': 'Interstellar', 'duration': 4000}
movie id not found
```

Raising exceptions review

Software applications aren't perfect, and user input and network connectivity are far from predictable. Despite intensive debugging and unit testing, applications will still have failure cases.

Loss of network connectivity, missing database rows, out of memory issues, and unexpected user inputs can all prevent an application from performing "normally". It is your job to catch and handle any and all exceptions gracefully so that your app keeps working. When you are able to detect that something is amiss, you should be raising the errors yourself, in addition to the "default" exceptions that the Python interpreter will raise.

```
raise Exception("something bad happened")
```

Different types of exceptions

We haven't covered classes and objects yet, which is what an `Exception` really is at its core. We'll go more into that in the object-oriented programming course that we have lined up for you next.

For now, what is important to understand is that there are different types of exceptions and that we can differentiate between them in our code.

Syntax

```
try:
    10/0
except ZeroDivisionError:
    print("0 division")
except Exception:
    print("different exception")

try:
    nums = [0, 1]
    print(nums[2])
except ZeroDivisionError:
    print("0 division")
except Exception:
    print("different exception")
```

Which will print:

```
0 division
different exception
```

Assignment

The `get_movie_record` function can now raise two different types of exceptions. One is an `IndexError`, the other is a custom exception message of the base `Exception` type. Complete the `handle_get_movie_record`. It should return the result of `get_movie_record` but if an `IndexError` is raised it will print `index too high`. Otherwise, if any other exception is raised it will just print the exception itself.

```
def handle_get_movie_record(movie_id):
    try:
        return get_movie_record(movie_id)
    except IndexError:
        print("index too high")
    except Exception as e:
        print(e)


# Don't edit below this line


def get_movie_record(movie_id):
    if movie_id < 0:
        raise Exception("negative ids not allowed")
    movies = [
        {"name": "Apollo 13", "duration": 128},
        {"name": "2001: A Space Odyssey", "duration": 300},
        {"name": "Interstellar", "duration": 4000},
    ]
    return movies[movie_id]
```

```
OUTPUT
{'name': 'Apollo 13', 'duration': 128}
{'name': '2001: A Space Odyssey', 'duration': 300}
{'name': 'Interstellar', 'duration': 4000}
index too high
None
negative ids not allowed
None
```

Raising exceptions review

As you've noticed, there are many kinds of exceptions. Many specific exceptions are built in to the language like `IndexError` and `ZeroDivisionError`, and all Exceptions count as the parent `Exception` type.

If you're interested in the official documentation on all the built-in exceptions you can find a list here.