Introduction to Python Documentation

Release 0.42

Raghuram Devarakonda

May 31, 2020

Contents:

CHAPTER 1

Introduction

This document contains material used in teaching Python to few middle school students.

In order to follow along, you need access to Python. One option is to use Python that is available on your computer (it usually comes preinstalled on Linux and perhaps even Mac) or install it yourself. But there is another far easier option which is handy if you only have access to Chromebook (as most middle school students do these days). That option is to use Python in the cloud and in particular this one:

> https://www.pythonanywhere.com/

The service supports free accounts which should be sufficient for the most part.

1.1 Quickstart guide for "pythonanywhere"

- Create an account. You can start with "free" tier.
- Go to "Dashboard".

The easiest way to explore is to use "consoles" by clicking on `>>> Python` and selecting a version. For the purpose of this course, we will use version "3.6".

For writing code in a file, click on `Files` from the main page and you will see how to open a new file. **Do remember to name your files with ".py" extension**. This will give you syntax highlighting. The script file can be run by clicking on `>>> Run`.

Finally, there is a more advanced option that you will need to use once you start writing more complicated scripts. From the dashboard, click on `$ Bash`. This will open a bash console where you can edit and run scripts directly. More information about this option will be provided later.

1.2 Resources

Main web page for Python: https://www.python.org/

Python tutorial: https://docs.python.org/3/tutorial/index.html

Documentation for Python 3: https://docs.python.org/3/

CHAPTER 2

Lesson notes

2.1 Lesson 1 - Numbers, Strings

Welcome to the very first lesson on Python. I am sure you are going to have a very nice time learning to program.

Python is a very popular programming language that is used in a wide variety of applications, games, and in building web sites. In this course, we will only focus on the very basic features of the language as the main emphasis here is not to master the language but rather to understand general programming concepts. Keep in mind that many things you learn here apply to other programming languages as well.

2.1.1 Data Types

When you start learning any programming language, the very first thing you should do is to understand what types of data that it can work with and what kind of operations that it supports on them. This is generally referred to as *types* or *data types*. The most basic data types that all languages support are numbers and strings so we will start with those.

Python supports all kinds of numbers that you know from your Maths class.

```
>>> 42
42

>>> 10.0
10.0
```

Python supports all the usual arithmetic operations on numbers so you can use it as a calculator:

```
>>> 42 + 10
52
>>> 42  * 10
420
>>> 42/2
21.0
>>> 42 - 10
```

(continues on next page)

(continued from previous page)

```
32

# Here is how you can calculate remainders.
>>> 42 % 10
2

# to calculate powers - called "exponential" operator
>>> 2 ** 3
8
```

BTW, did you notice how you can write comments in the code using # symbol? Any text following this character is ignored by Python so you can use it to write comments in your code (to explain what your code is doing if it is not already clear).

Let us now move to "strings". You will need to manipulate strings in many programs so it is very useful to have good knowledge about them. A string is a sequence of characters. A single character is also a string which has a length of 1. In Python, strings are enclosed in either double quotes or single quotes.

```
>>> "test string"
'test string'

>>> "c"
'c'
```

Python supports several operations on Strings. Here are some of them.:

```
# to convert a string to upper case
>>> "test".upper()
'TEST'

# to convert to lower case
>>> "TEST".lower()
'test'

# to replace some parts of the string
>>> "abcda".replace("a", "x")
'xbcdx'
>>> "aabbcc".replace("aa", "xx")
'xxbbcc'

>>> "test".capitalize()
'Test'
```

You can combine two strings using + operator.:

```
>>> "test " + "string"
'test string'
```

Note that the first string contains a "space" which is no different from any other character.

2.1.2 print() function

You can print various values using the function `print()`. You can either print single values or a combination of them.:

```
>>> print(42)
42

>>> print("test string")
'test string'

>>> print("My age is", 11)
'my age is 11'

>>> print("I am", 11, "years old")
I am 11 years old
```

Notice how Python is adding a “space” between items that are passed to “print”.

As you have already seen, you use a function by passing it some values in parenthesis. However, passing values is not mandatory (but you still need parenthesis). Here is an example:

```
>>> print()
```

Python has many other functions which we will learn in later lessons.

2.1.3 Variables

Many times in your program, you need to store values before using them later. For this, you use “variables”. Here is an example:

```
>>> age = 42
```

Here, we could have used “42” directly but instead, we created a variable called “age” which now contains the value “42”. You can now use “age” to mean 42 at any place in the code.

= is known as *assignment operator* which assigns values from right side to variables on the left side.

You can choose any name you want for variables (subject to some rules) but it is very important that you name them appropriately. **In particular, variables should be named such that they describe the values they may contain.**. This helps you and others in understanding the code, especially when you are reading it at a later time.

Apart from naming variables descriptively, you should not use Python function names to name your variables.

2.1.4 Formatting strings

You can combine strings and integers and even other data types in any format you want to form a new string. This is especially useful if you want to print some information to the console.

Here is an example:

```
>>> teststring = "This year: {}, Month: {}, Date: {}".format(2019, 2, 22)
>>> print(teststring)
'This year: 2019, Month: 2, Date: 22'
```

Here, `{}` are just place holders. They will be replaced by the values you pass to `format()` function.

2.2 Lesson 2 - Lists and if statements

In the last lesson, we learned basic data types - numbers and strings. In this lesson, we will study a new data type called *List*. We will also learn about some more operations that Python provides for these data types.

As you have already seen, it is useful to store values in variables so that our code is more readable. Example:

```
>>> count = 10
```

Now, if we want to increment this counter or add a number to it, this is how we do it:

```
>>> count = count + 1
>>> count = count + 10
>>> print(counter)
21
```

In addition to `print()` function, Python supports many other functions called *built-in functions*. These are called *built-in* because they are provided by the language itself. As you will learn later, you can also write functions and in fact, that is how you will organize your code for the most part. For a full list of built-in functions, see here.

2.2.1 Lists

Let us now get back to learning about data types supported by Python. Here we learn about "list" type. A list is a collection of other types. For example: a list of strings or a list of integers. A list is formed by enclosing the items with "square brackets", like so:

```
>>> ["red", "green", "blue"]
```

Let us use a variable to contain a list and then, we can see some list operations.

```
>>> colors = ["red", "green", "blue"]

>>> print(colors)
['red', 'green', 'blue']

# To add items to a list at the end.
>>> colors.append("magenta")
>>> print(colors)
['red', 'green', 'blue', 'magenta']

# To remove an item
>>> colors.remove("red")
>>> print(colors)
['green', 'blue', 'magenta']
```

Notice how the variable "colors" contains a list and various list operations can be performed using the variable.

A list contains elements of different data types. For example, the following list contains both strings and integers.:

```
>>> testlist = ["abc", "def", 1, 2]
```

Even though this is possible, most of the times, a list contains elements all of the same type.

Here are some more operations possible on lists.

To sort the list alphabetically:

```
>> colors.sort()
>>> print(colors)
['blue', 'green', 'magenta']
```

To reverse a list::

```
>> colors.reverse()
>>> print(colors)
['magenta', 'blue', 'green']
```

Notice how “sort” and “reverse” operations changed the data stored in the variable “colors”. What if we don’t want to affect the data but want to get a new list that is sorted or reversed?

```
>>> print(colors)
['magenta', 'blue', 'green']

# This gives a new list, leaving the original list unaffected.
>>> sorted(colors)
['blue', 'green', 'magenta']
>>> print(colors)
['magenta', 'blue', 'green']

>>> reversed(colors)
['blue', 'green', 'magenta']
>>> print(colors)
['magenta', 'blue', 'green']
```

2.2.2 len()

You will use the built-in function `len()` to find the size of the lists as well as that of strings. Examples:

```
>>> len("test")
4
>>> colors = ["red", "green", "blue"]
>>> len(colors)
3

# Empty string
>>> len("")
0
```

2.2.3 Boolean expressions

Many times in a program, you need to do something depending on whether some condition is true or false. For example, this is how you check if a number is even:

```
>>> i = 5
>>> if i % 2 == 0:
...     print("even")
... else:
...     print("odd")
```

As you can see, you use `if-else` statements to do this. This statement contains a *condition* that can be *true* or *false*. In the above code, the condition is `i % 2 == 0`. Here, we are checking if the remainder when a number is divided by 2 is 0. Apart from the condition itself, there is a block of code that is run when the condition is true and another block of code that gets run when the condition is false. Did you notice **:** after the condition and after `else`? That is how we indicate a block of code to Python. Also note that the code block is indented by few spaces. It is very important to use same number of spaces for a code block. The convention is to use 4 spaces.

Now, there are may operators that you can use as part of conditions. Some examples:

```
>>> i > 4
>>> i < 4
>>> i == 4 # to check if value of "i" is 4.
```

To check if two strings are equal:

```
>>> s == "test"

>>> test_string = "abc"
>>> len(test_string) < 5
```

Note that = is used to assign a value while == is used to check for equality.

The operator `in` is used to see if an element is in a list:

```
>>> "red" in colors
```

BTW, "else" part of the "if" statement is optional. For example:

```
>>> i = 5
>>> if i % 2 == 0:
...    print("even")
```

In this case, nothing happens if the number is not even.

Finally, it is possible to check for multiple conditions using a different variation of `if` statement. In the following example, we are checking if a number `n` is

- less than 10
- between 10 and 20
- or above 20

```
>>> n = 10
>>> if n < 10:
...     print("less than 10")
... elif n < 20:
...     print("less than 20")
... else:
...     print("above 20")
...
less than 20
```

2.2.4 Accessing list elements

Sometimes, we need to access individual elements of a list.:

```
>>> colors = ["red", "green", "blue"]
>>> print(colors[0])
'red'
```

Here, we are accessing the first element of the list. Note that the counting of the elements in a list starts with "0". So to print all the elements of this list, here is one way:

```
>>> print(colors[0])
>>> print(colors[1])
>>> print(colors[2])
```

The number we use to access an element is called *index*. The index starts from 0 and goes all the way up to the one less than the size of the list.

You will get an error if you try to use an invalid index.:

```
>>> print(colors[3])

IndexError: list index out of range
```

It is also possible to access elements from the end of the list instead of from the start. For example, the following will give you the last element of a list:

```
>>> colors[-1]
>>> # To access second element from the last:
>>> colors[-2]
```

2.2.5 Assignment

There is a new object in the solar system that is found to be traveling at 40000 miles per hour.

Write a program that calculates the number of days it takes this object to travel from Sun to Earth. Your program should print the following when run:

```
It takes N days
```

where `N` is the value your program should calculate.

Note. This object is named Oumuamua and in reality, it doesn't travel from Sun to Earth in straight line.

2.3 Lesson 3 - Problem solving techniques

In the last lesson, we learned control flow statements such as `if-else` and `for`.

In this lesson, let us try to write a program for this problem:

> "Given a day, the program should print if it is a weekday or weekend."

```
$ python3 day_of_the_week.py monday
weekday

$ python3 day_of_the_week.py sunday
weekend
```

First, we need to accept the input from the command line.:

```
import sys

day = sys.argv[1]
```

Now, we need to find what type of day it is. The most straightforward way of doing it is checking the input against each day, like so:

```
if day == "monday":
    print("weekday")
elif day == "tuesday":
    print("weekday")
elif day == "wednesday":
    print("weekday")
elif day == "thursday":
    print("weekday")
elif day == "friday":
    print("weekday")
elif day == "saturday":
    print("weekend")
elif day == "sunday":
    print("weekend")
```

Notice, how multiple if conditions can be combined using `elif`.

The program works but there are some issues with it. First, code looks a bit cluttered. Second, what if you need to write a similar program but with a bigger list? You can't keep adding `elif` statements.

What we are trying to do here is to see if a given input (name of the day in this case) belongs to a group (such as "weekday" or "weekend"). Such problems can be solved by using lists. Since there are two types of days here for the purpose of this problem, let us define two lists.:

```
weekdays = ["monday", "tuesday", "wednesday", "thursday", "friday"]
weekend = ["saturday", "sunday"]
```

We can simply check if the input is in the first list or second list:

```
if day in weekdays:
    print("weekday")
elif day in weekend:
    print("weekend")
else:
    print("not a day")
```

Notice how the second version is much smaller and much easier to read. In general, you should try to use types such as lists (and others provided by Python which we will see later) to solve problems.

Also note how we added `else` clause at the very end. Now our program gives a proper error message if the input is not valid. This is another thing you need to keep in mind when you accept inputs. **Your code needs to be prepared to handle both valid and invalid inputs**. In the case of latter, the program should provide a good error message.

2.3.1 Problem solving techniques

- Divide the problem into smaller ones and solve them. Once smaller problems are solved, you just need to figure out how to combine them. This technique is called *divide and conquer*.
- Don't try to write a perfect working program in the first go. Instead, write something that works and slowly improve it.
- Don't try to write the program directly. First solve the problem for some example inputs and see how you are solving it (you can use paper and pencil for this). Then, try to convert the solution into a program.

2.3.2 Assignment

Given a word, your program should print number of vowels in the word, like so:

```
$ python3 count_vowels.py "test"
Number of vowels: 1

$ python3 count_vowels.py "it is snowing"
Number of vowels: 4

$ python3 count_vowels.py "xyz"
Number of vowels: 0
```

You will be using most of the concepts you learned in this lesson and the previous ones to solve this problem, so go over the notes if required.

2.4 Lesson 4 - Functions

We have been learning about various data types and operations on those types. In this lesson, we will focus on the organization of the code itself.

Till now, we have only seen very small programs that span only around 10 lines. But in practice, the programs are usually much larger extending to hundreds or even thousands of lines. In order to maintain such large code, various techniques are used and we will learn one such technique called *functions*.

2.4.1 Functions

You have been already using functions such as *len()* and *sorted()* in your programs so far. They are called *builtin* functions because Python language itself supports them. But there is nothing special about them. You can write your own functions and use them in your code. Here is an example:

```python
def add(a, b):
    return a + b
```

This is a simple function that takes two parameters and returns their sum. This code shows all the main components of a function:

- Name. Every function should have a name (with few exceptions that you don't need to worry about now).
- Input parameters. A function can take any number of input parameters. It is also possible that a function doesn't take any parameters at all.
- Body. A function has a body of statements that does some work.
- Return value. A function usually does some work and returns a value. It is possible to return more than one value but we will talk about that later.

  Note that a function can return any type and in some cases, it doesn't return any value at all.

Now, you can call this function any where in your code where you need to add two numbers. Here is a sample:

```python
import sys

num1 = int(sys.argv[1])
num2 = int(sys.argv[2])

```

(continues on next page)

(continued from previous page)

```
def add(a, b):
    return a + b

total = add(num1, num2)
print("Total =", total)
```

To run this program:

```
$ python3 add.py 1 2
```

Write a function that checks if a number is even or odd:

```
is_even(n) # returns True or False
```

Here is a function that reverses a string:

```
def reverse_string(s):
    return "".join(reversed(s))
```

2.4.2 Advantages of functions

As has already been mentioned, the main advantage of functions is that it helps in breaking code into small manageable units. But there is one more important advantage as well. Some times, you need to do same computation at many different places in the code. In these cases, you will write a function to do the computation and then just call the function at other places.

This way, the actual computation is present in only place. If you find a bug in the code later on, you only need to fix at one place.

As an example, once you have the function *add()* as shown above, you can now call it everywhere in your code where you need to add two numbers. In this case, the actual code is extremely simple but in many cases, the code in function will be larger.

2.4.3 Guidelines for writing functions

- Name functions descriptively.
- Keep all the functions at beginning of the file and write the main code after that.
- Functions should not have large number of parameters. If you ever end up writing a function that accepts large number of parameters, you can usually break it in to multiple functions where each function accepts small number of parameters.

2.4.4 Assignment

Write a program that accepts a word and then prints if it is palindrome or not. Examples:

```
$ python3 palindrome.py eve
is a palindrome.

$ python3 palindrome.py abc
is not a palindrome
```

Remember to use functions that we learned in this lesson.

2.5 Lesson 5 - Number of byes

2.5.1 Built-in function *int*

The built-in function *int* can be used to convert strings or floating point numbers to integers.

```
>>> int("10")
10
```

```
>>> int(5.123)
5
```

```
>>> int(5.9)
5
```

2.5.2 Number of byes in a tournament

We will try to write code to solve the following problem.

In a tennis tournament, how many players need to be given byes?

If the number of players who sign up for a tournament is not equal to power of 2, some players need to be given byes and they automatically move to second round. For example, if there are 21 people, 11 people will need to be given byes. More details about this problem can be found here:

http://draghuram.github.io/tournament/byes/2018/01/20/how-many-byes.html

Don't worry if the math on that page looks complicated. You don't need that for writing code. You just need to know the solution which is this:

Find a power of 2 that is greater than number of players and subtract number of players from that number.

For example, if there are 21 people in the tournament, 32 is the closest power of 2 that is greater than 21. So the number of byes would be:

```
32 - 21 = 11
```

This is all you need to know to start writing code.

2.5.3 Solution

To solve this problem, we need to learn a function called `log` in the module `math`. This function computes the *logarithm* of a *number* with respect to a *base*. Here are some examples:

```
>>> import math

>>> math.log(16, 2)
4.0
>>> math.log(8, 2)
3.0
>>> math.log(32, 2)
5.0
```

Can you understand the return value of `log()` function? Here is another way of understanding it:

```
>>> 2 ** 3 # 2 to the power of 3
8
>>> math.log(8, 2)
3.0

>>> 2 ** 4 # 2 to the power of 4
16
>>> math.log(16, 2)
4.0

>>> 2 ** 5 # 2 to the power of 5
32
>>> math.log(32, 2)
5.0
```

As you can see, `log()` is kind of opposite to `power` operator. Armed with this new function, let us proceed with the code to solve our problem. Remember that this is the algorithm to solve our "byes" problem:

Find a power of 2 that is greater than number of players and subtract number of players from that number.

Let us assume that there are 21 players in a tournament. Our job is to find the number of byes that need to be given.

```
>>> math.log(21, 2)
4.392317422778761
```

Note that we are getting a floating point number here because 21 is not an exact power of 2 number. We need to convert this to an integer first.

```
>>> int(4.392317422778761)
4
```

Note that 2 to the power of 4 will give us a power of 2 number that is less than 21. But what we really want is a power of 2 that is greater than 21. This is how we do it.

```
>>> 2 ** (4 + 1)
32
```

All we need to do now is to subtract number of players from this number.

```
>>> byes = 32 - 21
```

There you go! We now have code to solve the problem when the number of players is 21.

2.5.4 Assignment

Write a program that accepts number of players as input (integer) and prints number of byes that need to be given. You need to take code explained above and generalize it. Try to write the code using functions.:

```
$ python3 num_byes.py 21
11

$ python3 num_byes.py 10
6
```

Bonus points if you can write the program so that when a power of 2 is given as input, the output is 0 (because there is no need to give byes in this case).:

```
$ python3 num_byes.py 16
0

$ python3 num_byes.py 128
0
```

CHAPTER 3

Concepts

3.1 Operating System Concepts

It is very useful to learn some operating system level concepts as they will help you in understanding Python programs better.

An operating system is a type of program that runs directly on the hardware and makes other applications like Python possible to run. Some examples of OS include: *Linux*, *Windows*, *MacOS*, *iOS*, *Android* etc.

3.1.1 Shells

When you login into a machine, a program called *shell* is started. The most popular shell on *Linux* is called *bash*. Shells primarily do the following in an infinite loop:

- Wait for user to enter a command
- Run the command

For example, on PythonAnywhere, a kind of Linux machine is created for each user and when you get a *bash console*, you are logging into this machine and *bash* is started for you where you can enter commands.

3.1.2 Some useful Commands

- `ls` - lists files and folders
- `cp` - copies files and folders
- `mv` - renames files and folders
- `rm` - removes files and folders
- `ps` - shows running processes

3.1.3 Command components

- A command (or program) has mainly three components:
  - program name
  - options
  - arguments or parameters
- You can access all this information in your python program.

For example, you can access command line parameters using the list `sys.argv`.

3.1.4 Command options

- All commands support options that change their behavior.
- Examples
  - *ls -l*, *ls -ltr*. Lists detailed information for files.
  - *cp -i*. Warns you if you are overwriting a file.
  - *mv -i*, *rm -i*. Same as above.
  - *rm -r*. Removes directories and any files with in those directories as well.

3.1.5 Command line parameters

- You can pass information to programs using command line parameters.
- If the values have spaces in them, place single or double quotes around the values.

3.1.6 Editors

- Editors are programs that you use to create and modify files.
- There are hundreds of editors available. Some examples:
  - *vi*, *emacs*
  - editor on PythonAnywhere site
- Editors support *syntax highlighting* where different language features are shown in different colors.

CHAPTER 4

Problems

4.1 Guidelines

Please follow these principles while writing programs.

- Name variables descriptively. One should be able to understand the purpose of the variable by its name. Same principle applies to other things such as program names, function names etc.
- Write comments in your code if you think the code is not easy to understand. comments start with # character. Here is an example:

  ```
  # Split the match score string to get scores for each set.
  set_scores = match_score.split(" ")
  ```

  At the same time, your first priority is to write code that is easy to understand without comments. Only when the intent of the code is difficult to follow, should you write the comments.
- Your program should gracefully handle error conditions as much as possible. You should always assume that the input given to the program may be invalid and should try to print an error message in that case.
- Break down the problem into smaller sub-problems and try to solve them independently. Use *functions* where ever you can.
- Test your program thoroughly. Test with valid inputs as well as invalid ones.

4.2 Who won the match?

Given the score for a tennis match, your program should print which player (player 1 or player 2) won the match. Here is an example run:

```
$ python3 tennis_scores.py "6-4 6-4"
Player 1 won the match
```

(continues on next page)

(continued from previous page)

```
$ python3 tennis_scores.py "4-6 4-6"
Player 2 won the match

$ python3 tennis_scores.py "4-6 6-4 4-6"
Player 2 won the match

$ python3 tennis_scores.py "6-4 5-7 6-4"
Player 1 won the match
```

You can assume that the scores will be given as a single string as shown above so you need to split the string to get scores for individual sets. You need to further split the set score to find the games won by each player.

Even though examples above show scores for best of 3 sets match, your program should work for best of 5 sets as well, such as this:

```
$ python3 tennis_scores.py "6-4 5-7 6-4 7-6"
Player 1 won the match
```

Bonus points if you can handle invalid input and print error message as follows:

```
$ python3 tennis_scores.py "6-4 5-7 6-4 6-7"
Invalid input
```

4.3 Special SSN

Given a 9 digit number, check if it follows this pattern:

- First digit should be divisble by 1.
- First and second digits together should be divisble by 2.
- And so on. Full number should be divisble by 9.

Here is an example: 123252561.

Here "1" is divisble by 1, "12" is divisble by 2 etc.

Example run:

```
$ python3 special_ssn.py 123252561
123252561 follows special SSN pattern.

$ python3 special_ssn.py 123252562
123252562 doesn't follow special SSN pattern.
```

If the length of the input number is not exactly 9, your program should print an error message. Like so:

```
$ python3 special_ssn.py 123252
input is invalid.
```

Once you get this working, let us make the problem more generic. Instead of 9 digits, your program should accept any number of digits and check if it follows the pattern. Examples (all the following examples numbers follow the pattern):

```
$ python3 special_ssn.py 1

$ python3 special_ssn.py 12

$ python3 special_ssn.py 123

$ python3 special_ssn.py 1232

$ python3 special_ssn.py 12325

$ python3 special_ssn.py 123252

$ python3 special_ssn.py 1232525

$ python3 special_ssn.py 12325256

$ python3 special_ssn.py 123252561
```

4.4 Music Player

Raspberry Pi has a program called “omxplayer” that can play MP3 files (in addition to other formats). Without any options, the program starts playing the track at the beginning. Here is an example.

```
$ omxplayer track1.mp3
```

The command has an option called `--pos` that can be used to start playing at any arbitrary position in the track. For example, the following command will start playing from 30 minute mark.

```
$ omxplayer --pos 00:30:00 track1.mp3
```

Now, if you kill the command and re-run it, it will not remember where it stopped but instead, it will start playing either at beginning (or at the time passed using `--pos` option).

Your task is to implement a program called `musicplayer.py` that takes a MP3 file as input and plays it.

```
$ musicplayer.py track1.mp3
```

If you kill the program and run it again, it should resume playing the track where it left off (a few seconds rewind is ok). So you need to come up with a mechanism of “remembering” the play location.

Once you have this program working, extend it so that it can play multiple tracks. The input is a file which contains multiple tracks, with one track per line. Here is a sample input file:

```
$ cat playlist.txt

track1.mp3
track2.mp3
track3.mp3

$ musicplayer2.py playlist.txt
```

In the second form, the program will play all the tracks in the play list file, in order. But again, it should remember the location so that when it is killed and rerun, the player should start at the right track and at the right position in that track.

With a program such as `musicplayer2.py`, we can use Raspberry Pi as sort of an iPod where it can start playing a playlist every time it is turned on.

CHAPTER 5

Indices and tables

- search