Ivelin Demirov

LEARN PYTHON 3.0 VISUALLY

An accelerated learning method which uses science and creativity to teach the right brain non-coders

Thank You!

A book takes many people working together to make it a reality. So, to all the Kickstarter backers that helped me complete this project I wish to express a big 'Thank You' and I hope we can do more great things in the future.

Anthony Storms Akins

Alan Prak

Alessandro

Andy Lee

Bill Stull

Brandon Thompson

Charles Rogers

Clément Labadie

Demian Vonder Kuhlen

Edgar Rodriguez

Gordon

Greg_Vee

Gregory Pfrommer

Gregory R. Kinnicutt

J.T. O'Gara

Janakan

Jeff

Joe

Joshua Bauder

Marco Randall

Marshall Walker

Neer Patel

Nigel J Allen

Padrao

Perry Yap

R.C. Lewis
Randy Morris
Rowan Knight
Sebastian Lange
Shahid Ali Shah
Shrawan
Stan Spears

Credits & Legal

AUTHOR
Ivelin Demirov

EDITORS
Anton Berezin

Beth McMillan

Page design

Jordan Milev

ILLUSTRATOR
Ivelin Demirov

PROOFREADER
Beth McMillan

ACKNOWLEDGMENT
I must acknowledge the help of the online Python community, who have toiled in the background for many years to help make Python the exciting programming language it has become

PAPERBACK ISBN
ISBN-13: 978-1507727072

ISBN-10: 1507727070

CONTACTS
pythonvisually.com

Contents

Alternative branches
Classes and objects
Collections
Tuples
Lists
Dictionaries
Loops
While
Loops
File operations
Exceptions
Custom exceptions
Import statement and Python modules and packages
Import statement
File operations using os module
File operations using os module
Regular Expressions
Regular Expressions
Math module
JSON
SQL: sqlite3
Python sqlite3
sqlite3
3rd party packages
Web Development with Flask

About this book

THE PROBLEM

People cannot be blamed for thinking programming is hard when trying it out for the first time. Learning to program is like learning a new language.

A number of rules and grammar syntax guidelines exist to follow. It also requires memorizing a bevy of glossary terms for each language, and unless a person works with programming at least 8 hours a day, the person is unlikely to become very familiar with programming quickly; at least, that has been the situation for years until now.

THE METHODOLOGY

Our approach involves teaching programming concepts via simple illustrations.

The visual approach works because it is one of the most fundamental ways of learning. Everyone as a baby and toddler learns the world around them via sight and sound long before there is comprehension associated with letters and meanings.

Programming works in modules and building blocks. As a person learns a few basic modules and steps, he can then learn to build more complex modules on those first basic units.

It's a building block approach where bigger structures can be coded once the basic modules are mastered. I start with a set of basic building blocks that are easy to learn through illustrations and metaphors.

From there, a user can apply multiple variations and build further. However, the initial set of building blocks becomes a Rosetta stone of sorts, allowing a user to program and build in any situation going forward.

Why Python?

When you embark on your computer science journey at college, you'll probably start by learning Python.

So why choose Python over any other programming language?
It's simple:

- Regardless of your operating system, Python is easy to set up.
- Python code uses plain English, making it easy to understand.
- Python will let you run bad code.
  Why is this good?
  When you're learning you're bound to make mistakes.
  Python tells you what the error is after you run the code, helping you to find the problem.
- Python is a very powerful tool, but it's also easy and fast to learn, which is why most people prefer it.

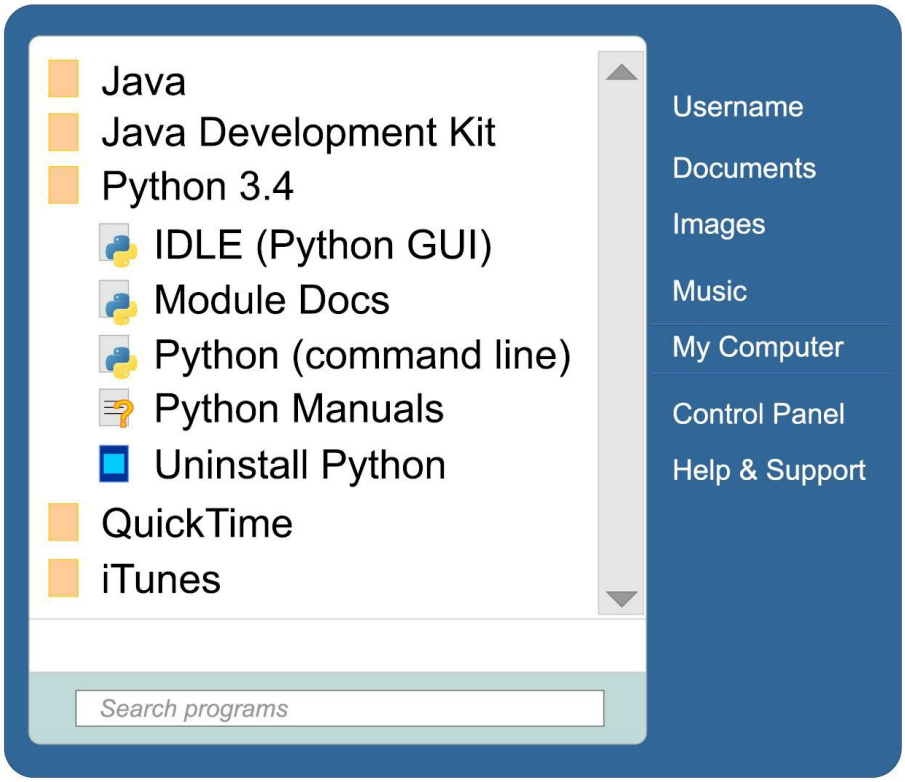
Installation

There are pre-compiled packages made for easy installation on multiple operating systems such as Windows, Linux, UNIX and Mac OS to name but a few.
If you head over to https://www.python.org/downloads/ you can select the right installation method for your operating system. Most Linux systems come with Python pre-installed.

Version 3 of Python is used in this book.
In Windows, once you have installed Python, select Python IDLE or Python command line/console from the Start menu or screen.

The difference between the IDLE and the console is that the IDLE has a graphical interface, which looks similar to a text editor. In Linux, UNIX, and OS X, you can launch Python emulator from command line by typing Python.

To choose a specific version of Python type PythonX where X is the version number (e.g. "Python3" for version 3).

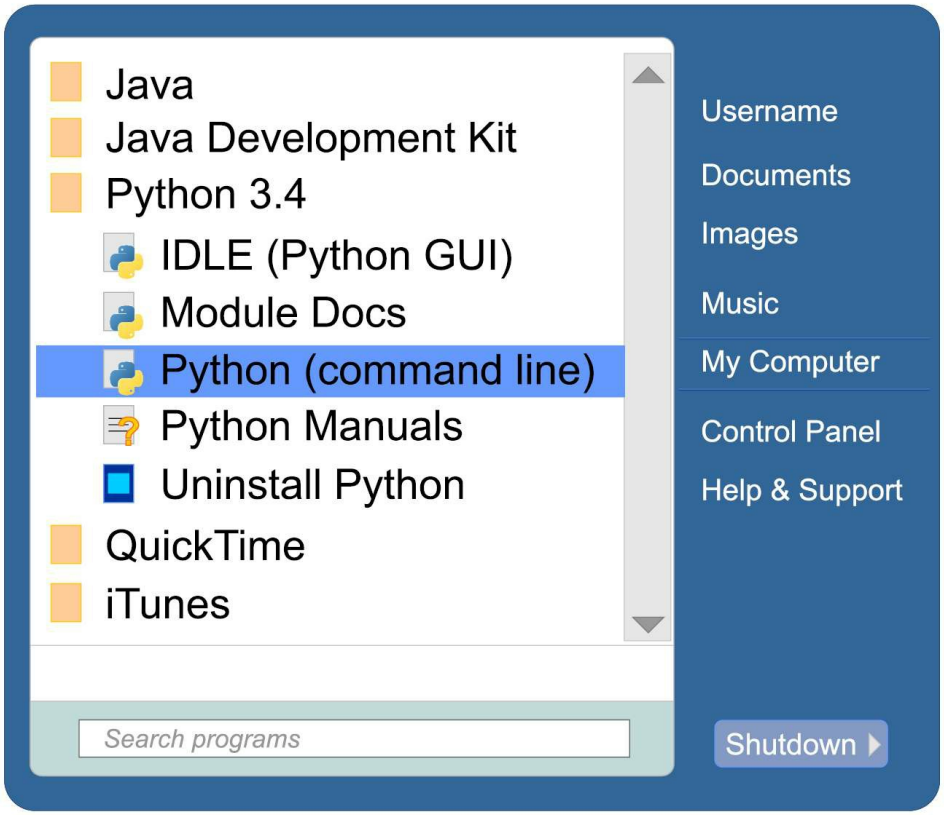


Hello World!

Most of the time that you write Python code, you will be writing the script in the IDLE or any other IDE like Eclipse or rich text editor like Sublime text or Notepad++. However, you can use the Python interpreter to write code interactively as it acts like a UNIX shell,

Even though it is possible to create programs using just the interpreter it is strongly recommended not to do so since it is hard to save the code as a script for further reuse. Rather consider the interpreter as an "on the fly" testing tool for your coding ideas.

Now lets make the interpreter output "Hello World!" string on the screen. Lets do it via the interpreter.

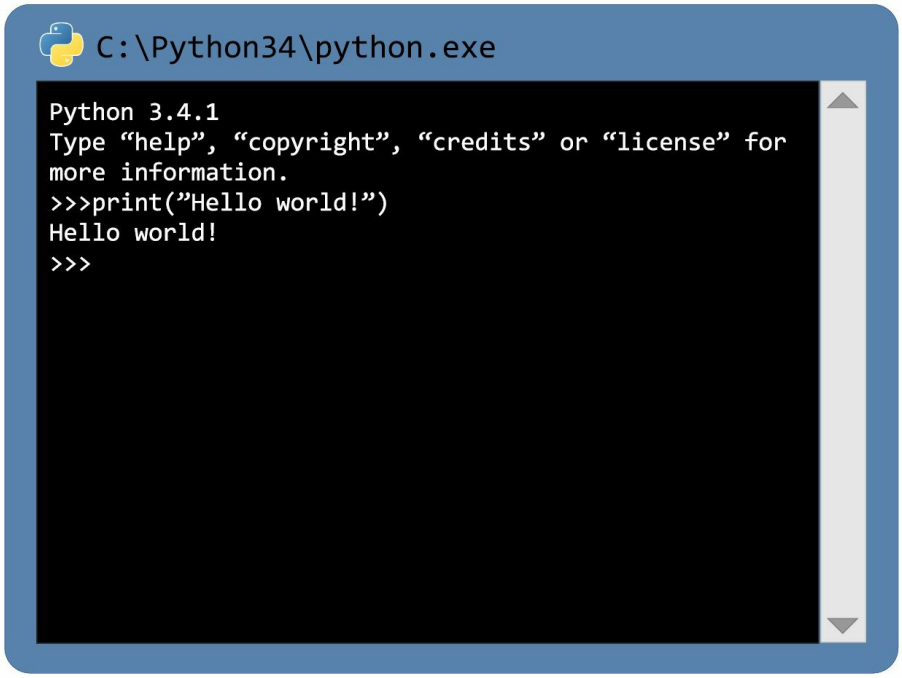


Terminal way

Launch the interpreter from command line using Python command as it was described in the previous chapter.

Type the following: print("Hello world!") and hit "Enter".

The terminal shall look the following way:



In the code snippet above you can see that the first line starts with >>>.
This symbol indicates the line where you provide dynamic input to the interpreter. Notice, as soon as the result of your first command was printed on the screen the interpreter printed >>> again. Giving instructions to the interpreter is sequential. You give one instruction - you wait till execution finishes, you enter another command and so on.

Note, further on in this book if you spot >>> symbol in a code snippet it shall mean that you are expected to test the code in the interpreter yourself.

On the first line you entered a statement. The statement is nothing else but an instruction for a computer to perform some task. The task could be anything from make_a_coffee to launch_a_rocket. In your case it is a simple print function though. Functions will be covered further in this book.

Scripting Way

One of the main principles of software engineering is reusability.
Let's make the code that was just tested in the interpreter reusable.

To do so, create a file called "hello_world.py" and type the following text inside:

```
1. print("Hello World!")
```

What you placed into the file is exactly the line that was previously executed via the interpreter.

Now lets execute the script that you wrote.

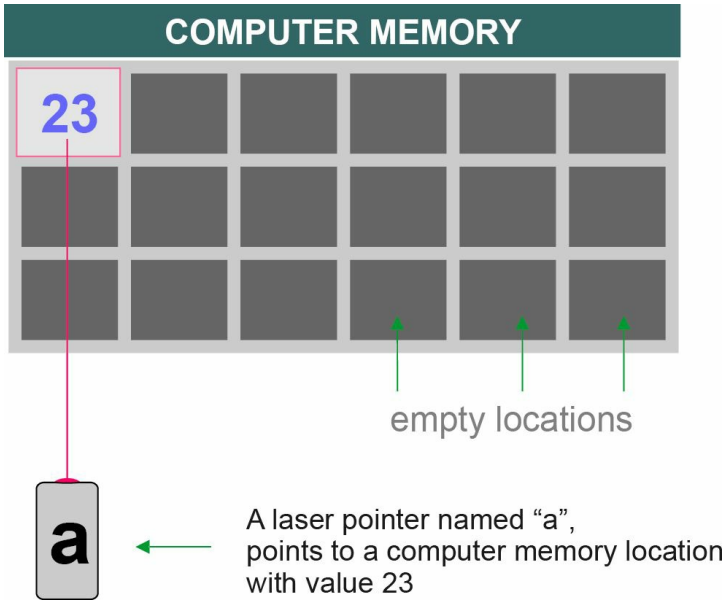
Open the command line, navigate to the location where the script resides and type the following: python hello_world.py. After you hit enter you shall see that a "Hello World!" text was placed on the screen.

Congratulations!
You've managed to write your first Python program and successfully launch it.

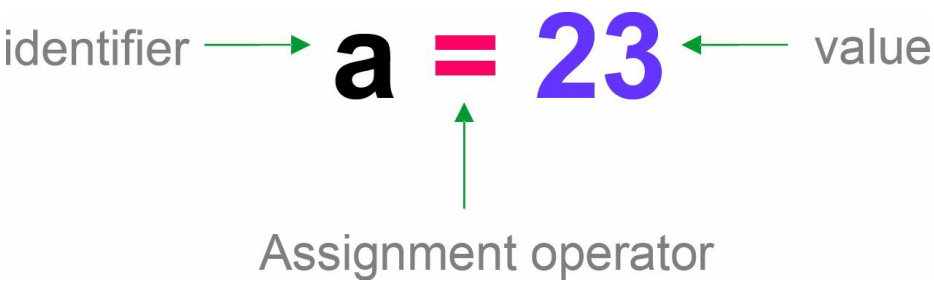
Part I: Built-in language features

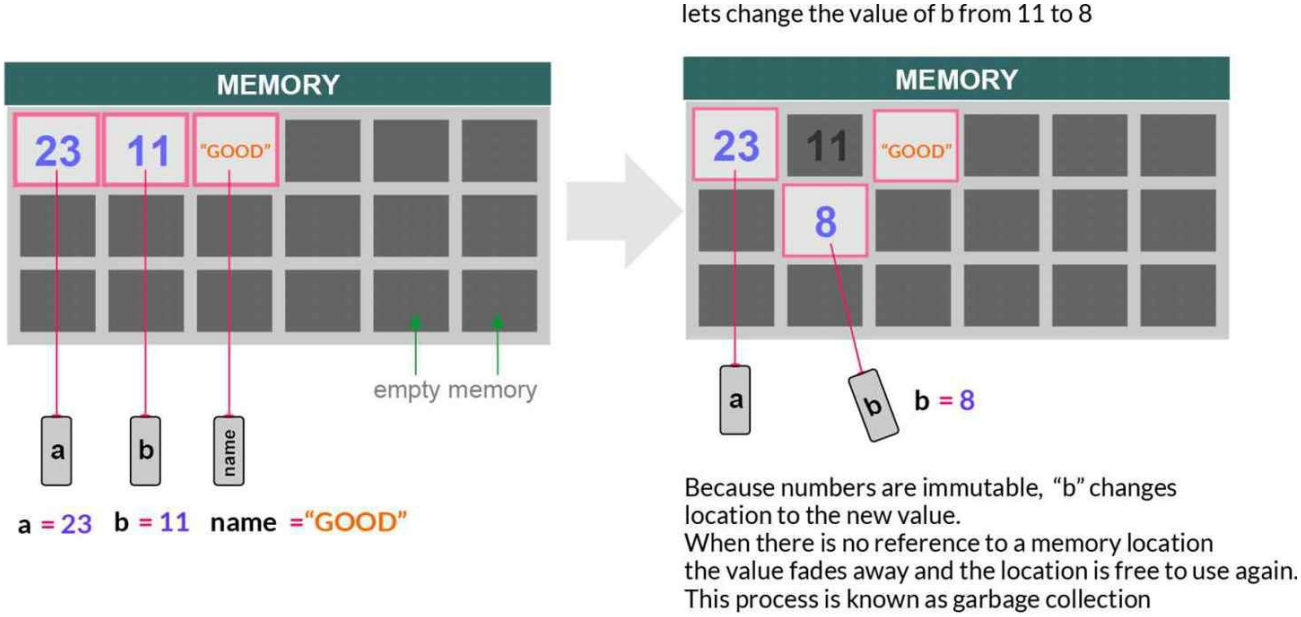
Variables

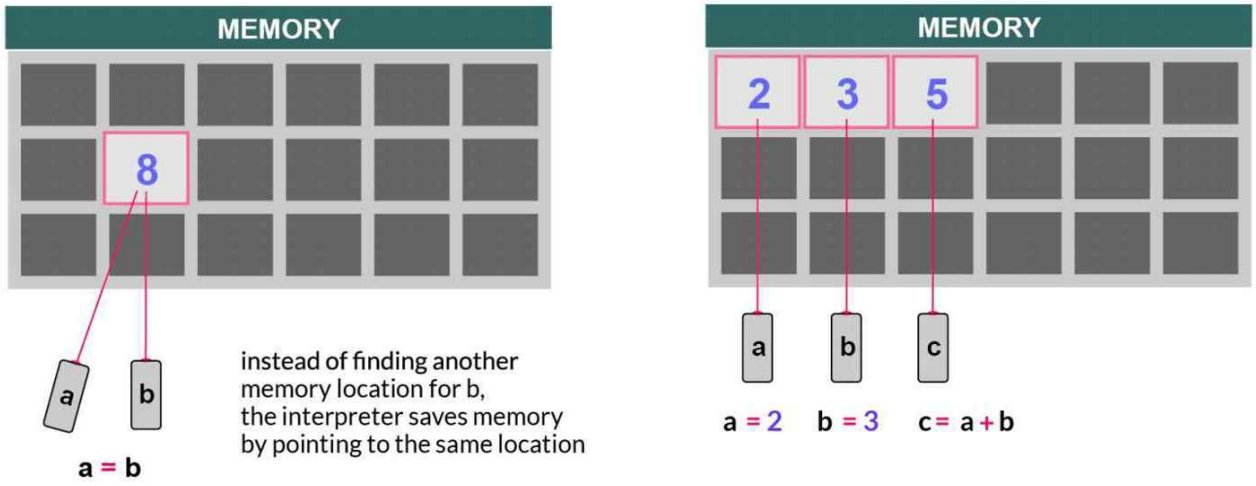
In Python, a variable can be used to store the output of a statement in the computer's memory.



The equals sign is used to assign a value to a variable.
The name of the variable is called its identifier







This is how to test if '**a**' and '**b**' share the same memory location

```
>>> a is b
True
>>> print (id(a))
123123123
>>> print (id(b))
123123123
```

You may assign the same value to multiple variables using the following syntax:

```
>>> a = b = c = d = f = "Hello World!"
>>> a
Hello World!
>>> c
Hello World!
>>> f
Hello World!
```

Python variable names must follow the following rules:

1. They must start with a letter or an underscore
2. They must contain only letters, digits or underscores

Several examples of illegal variable names:

```
1. 2var
2. var-bar
3. foo.loo
4. @var
5. $var
```

Python also has some special words that can't be used as variable names:

```
and   assert   break  class  continue def   del    elif
else  except   exec   finally for     from  global  if
import in      is     lambda not     or    pass    raise
return try     while  yield
```

Exercises

PROBLEM: Which statements are valid?

```
1. $foo = "bar"
2. lambda = "bar"
3. finally = "yes"
4. pass = "you shall not"
5. ___buga_wuga_ = "@@@huz"
6. foo_bar = "bar"
```

ANSWER: The two last ones. Note, the statement before the last one just looks incorrect. In reality it is a completely legal variable with a proper string value

Exercises

PROBLEM: Assign value "blah" to variables zen, foo and bar in one line.

ANSWER:

```
foo = bar = zen = "blah"
```

Data types

The following table provides an overview of the most frequently used data types in Python.

Variable type	Example	Usage comment
bool	`life_is_good = True` `hamsters_are_evil = False`	true/false values

int, long	`size_of_shoes = 42` `earth_population = 7000000000`	various whole digits
float	`pi = 3.14159265359`	not whole digits - with one or more signs after a dot
str	`chinese_hi = "你好"`	any text
None	`my_new_book = None`	empty variable without any meaningful value

You can store data of all types inside a variable using the assignment operator "=".

Multiline strings can be used to store large text blocks.

```
long_text = """ Line one
Line two
Line three
Line Four
"""
```

As you can see, a multiline string is a normal string enclosed by triple quotes instead of single ones.

In Python it is possible to convert strings to integers and integers to strings:

```
>>> str(100)
'100'
>>> int("234")
234
```

Quiz

PROBLEM:

1. What is the name of this = operator?
2. How to convert a digit 3.14 into a string '3.14'?

ANSWER:

1. assignment operator
2. str(3.14)

Basic Math

Python has a bunch of built-in arithmetic operators. The table below provides a brief comparison of the short notations and their longer equivalents.

Assume that initially the following is true: `a = 3`.

Operation name	Short notation	Long notation	Value of a	Comment
Addition	`a += 1`	`a = a + 1`	4	
Subtraction	`a -= 1`	`a = a - 1`	2	
Multiplication	`a *= 2`	`a = a * 2`	6	
Division	`a /= 2`	`a = a / 2`	1.5	Returns decimal values or float numbers
Modulo	`a %= 2`	`a = a % 2`	1	Returns an integer remainder of a division
Exponent	`a **= 2`	`a = a ** 2`	9	Similar to a^2 in regular maths
Floor division	`a //= 2`	`a = a // 2`	1	Returns only decimal (int) values
Negation	`a = -a`	`a = 0 -a`	-3	Returns the same value with an opposite sign

Finding the square root of a number is also easy in Python, but it's not a built-in language feature. You'll find out about it later in the book!

You can calculate the absolute value of a digit (e.g. |-3|), using abs(). For example, abs(-3) returns 3.

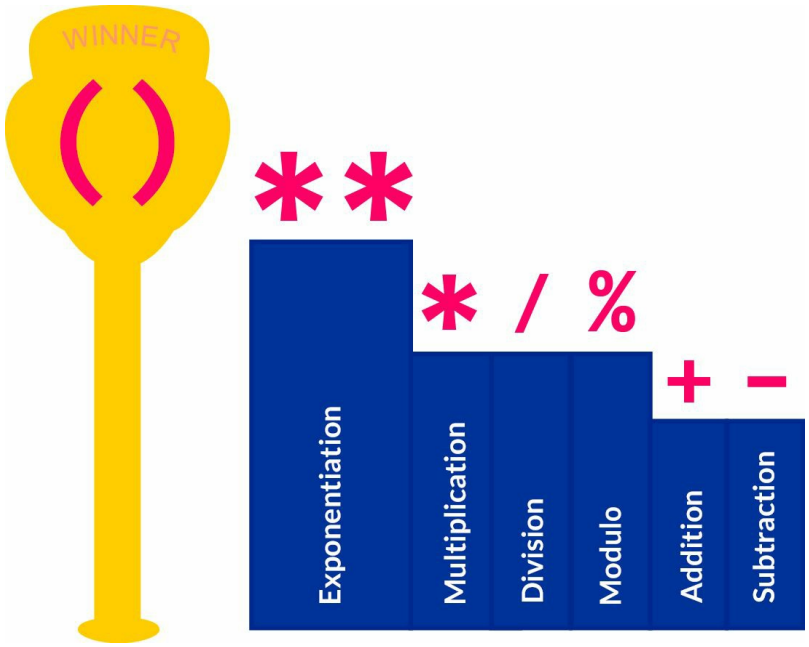
Operators precedence

The order of execution of mathematical operations in Python is similar to the order used in conventional mathematics.

In maths there are three general operator priority levels:

1. exponents and roots
2. multiplication, division and modulus
3. addition and subtraction

Exponents and roots are represented by functions of a standard library in Python and are covered further on in this book.



All other priority levels have their own built-in Python operators.

Higher order operations are performed before the lower ones. All the operations of the same order are executed one by one from left to right.

For example, the statement: 2 + 3 * 4 - 1 returns 13. The multiplication is performed before the addition and subtraction.

In order to affect the order of operator execution, you should use parentheses (). For example, (3 + 6) * 2 equals 18 and 8 / (1 + 1) equals 4.

Code Samples

The code snippet below shows how simple maths operations are processed in the Python interpreter.

```
>>> 13 * 2
26
>>> 10 - 32
-22
>>> 11 % 10
1
>>> 6 ** 2
36
>>> 12 / 8
1.5
>>> 12 % 8 + (12 // 8) * 8
12
>>> 6 - -16
22
>>> a = 16
>>> a += 3
>>> a
19
>>> a = 4
>>> b = 6
>>> a *= b
>>> a
24
```

Exercises

PROBLEM: Try to answer the following questions without Python and then use the interpreter to verify your answers:

1. What is the result of the following calculation: 2 + 2 + 2 + 2 + 2 * 0?
2. What about this one: (2 + 2 + 3 + 4) * (4 * 4 - (2 * 6 + 4))?

ANSWER: 10 - if you answered 0 you forgot that multiplication has a higher precedence than addition

1. 0 - if you answer was something else you forgot that operations in brackets should be performed first - a product of the second expression in brackets - (4 * 4 - (2 * 6 + 4)) is 0 and anything multiplied by 0 is 0 as well

Exercises

PROBLEM: A car has a speed of 100 km/h. Write some code that calculates the distance in meters (not kilometers!) that the car would travel in N hours. Set N to 10.

ANSWER:

```
1. N = 10
2. distance_in_kilometers = 100 * N
3. distance_in_meters = distance_in_kilometers * 1000
```

Exercises

PROBLEM: Rewrite the code using short notations

```
1. N = N * J
2. i = i + j * 18
3. f = f - k / 2
```

ANSWER:

```
1. N *= J
2. i += j * 18
3. f -= k / 2
```

Exercises

PROBLEM: Given that f(x) = |x^3 - 20|, find f(-3). Your answer should consist of two lines of code: one which initialises x to -3, and one that calculates the result.

ANSWER: The following Python code shall do the job:

```
>>> x = -3
>>> abs(x ** 3 - 20)
47
```

Exercises

PROBLEM: Try to solve the previous problem using -3 inline, rather than using the x variable.

ANSWER:

Your solution should look like this:

```
>>> abs((-3) ** 3 - 20)
47
```

If it looks like this:

```
>>> abs(-3 ** 3 - 20)
47
```

it is wrong because negation (i.e. subtraction from zero) has lower precedence than the exponent and thus has to be enclosed in parentheses. Even though your solution gives the same result as the correct one, it is a pure coincidence. Try to find value of x for f(x) = |x^2 - 20| with and without parentheses around the negation to see the difference.

Exercises

PROBLEM: How can you check if an integer is odd using Python?

ANSWER: Use the modulo operator to find the remainder when you divide by 2.
If the remainder is 1, the digit is odd.

```
>>> 3 % 2
1
>>> 4 % 2
0
```

String operations

Words and sentences are collections of characters, and so are Python strings.

You can access any character in a Python string using its index, either counting from the beginning or the end of the string. When you start from the beginning, you count from 0.

name = “My Name is Mike”

INDEXING	M	y		N	a	m	e		i	s		M	i	k	e
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In Python integer indices are zero based.

```
>>> name = "My Name is Mike"
>>> name[0]
'M'
>>> name[1]
'y'
>>> name[9]
's'
>>> name[-1]
'e'
>>> name[-15]
'M'
>>> name[7]
' 's
```

Slicing

You can take shorter substrings from inside a longer string.
This operation is called 'slicing'.

```
>>> name[11:14]  # from 11th to 14th, 14th one is excluded
'Mik'
>>> name[11:15]  # from 11th to 15th, 15th one is excluded
'Mike'
```

If you want to check if a string contains a particular substring, you can use the 'in' operator.

```
>>> "B" in "Foo"
False
>>> "B" not in "Foo"
True
>>> "G" in "Gohl"
True
>>> "James" in "James is tall"
True
```

The len function calculates the length of a string.

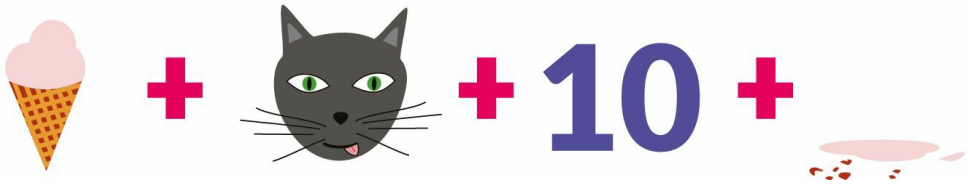
```
>>> len(name)
15
```

Concatenation

In Python it's possible to combine two strings into one:

The operation is called concatenation.

cone + cat + ten + ate




```
>>> boo = "cone" + "cat" + "ten" + "ate"
>>> print(boo)
conecattenate
>>> foo = "one"
>>> bar = "two"
>>> full = foo + " " + bar
>>> print(full)
one two
```

If you want a string containing N substrings, you use the * operator.

```
>>> foo = "foo " * 3
>>> print(foo)
foo foo foo
```

Quiz

PROBLEM:

1. In Python it is possible to combine multiple strings into one.
   What is the name of this operation?
2. There is a string value = “foobar”. You access a substring the following way:
   other_value = value[1:3].
   What is the name of this operation? How long is other_value?
3. What is the index of the first character in a string?

ANSWER:

1. Concatenation
2. Slicing. The length is 2. The last character defined by slicing operation is not included.
3. 0

Exercises

PROBLEM: There are two strings: one = “John is strong” two = “Tea is not warm”.
Write Python code to produce a string result = “John is not strong” out of one and two:

ANSWER:

```
result = one[0:8] + two[7:11] + one[8:14]
```

Exercises

PROBLEM: Find which of these strings doesn't contain the letter "B": "Foo", "Bar", "Zoo", "Loo".

ANSWER:

```
>>> "B" in "Foo"
False
>>> "B" in "Bar"
True
>>> "B" in "Zoo"
False
>>> "B" in "Loo"
False
```

Exercises

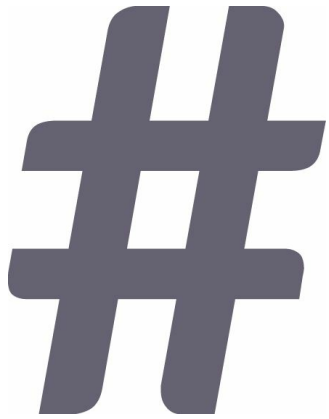
PROBLEM: Use the Python interpreter to find the length of the string “John is strong”

ANSWER:

```
>>> len(“John is strong”)
14
```

Comments

Writing comments on Python code is rather simple.
There are two ways to write Python comments, single line comments, and multiple line comments.



Comments are intended to help the user to understand the code, and are ignored by the interpreter.

Single line comments are created by adding the # character to the beginning. They are useful for adding small comments at the beginning of functions.

```
1. # This is a valid single line comment.
```

You can also add inline comments after a line of code, which can be useful for clarifying statements.

```
1. print("Hello there!")      # This would be an inline comment in Python
2. # This is another.
```

Multi-line Comments

Rather than chaining several single line comments together, for larger comments, you can use three consecutive quotation marks on the first and last line.

```
"""This is a better way
of using  comments that span
over multiple lines without
having to use lots of hashes.
"""
```

This type of commenting is often used as documentation for anyone reading the program. After all, no matter how skilled you are, if you get a program with no comments, you will have to spend time checking the code to figure out what it does and how to use it unless you knew beforehand.

Functions

We talked before about how reusability is a key principle of software engineering. Once you've written some code, you'll want to store it so you can use it again without having to copy and paste.

This is what functions are for! Functions are a way to wrap up your code so it can be used again by calling an easy shortcut.

Let's say you want to be able to print "Hello Mr. Jones! How are you today?" three times in a row without any code duplication. This can be achieved in the following way:

```
1. def hi():
2.     print("Hello Mr. Jones! How are you today?")
3. hi()
4. hi()
5. hi()
```

Basic functions

In the code snippet above you can see the keyword “def”. This defines a code block that represents the function. Function names have to follow the same rules as variable names. This function is called “hi”, and is followed by parentheses and a colon. Inside the parentheses is the argument list, which in this case is completely empty.
After the function, the code block has to be indented to define it as a nested block.

There are two Python conventions you should stick to: use four spaces to indent your code blocks, and use only lowercase letters and underscores in function names.

This code would not work:

<non-indented example>

```
1. def hi():
2. print(“Hello Mr. Jones! How are you today?”)
```

This code would not work:

<inconsistent indentation>

```
1. def hi():
2.         print(“Hello Mr. Jones! How are you today?”)
3.     print(“foobar”)
4. hi()
```

Below the function you may see three lines where it is invoked.
Lets have a look at one particular invocation:

The invocation is just a function name followed by the arguments that are supposed to be passed as an argument list. The arguments are expected to be placed within brackets. In this case we have no argument list thus the brackets are empty.

Functions with parameters

Sometimes, you'll want functions to alter their behaviour in different conditions. To do this, you can use function arguments.

Let's say you want to say hi not just to Mr. Jones, but to a person with any name. You'll have to edit the original hi() function to look like this:

```
1. def hi(name):
2.     print("Hello " + name + "! How are you today?")
```

The function now accepts a single argument, which is a variable called "name". Later, "name" is concatenated with two parts of the original string to give your new greeting.

Now you can say hi to John, James, and Maggie, like so:

```
1. hi("John")
2. hi("James")
3. hi("Maggie")
```

This would sequentially print:

```
Hello John! How are you today?
Hello James! How are you today?
Hello Maggie! How are you today?
```

Note, instead of passing a value directly you can assign the value to a variable and pass the variable to the function.

```
1. first_name = "John"
2. hi(first_name)
```

Once you've set up the function to use an argument, trying to call it without supplying an argument will cause an error.

Default values

To avoid these errors, you can give your arguments default values, which can be overridden if needed.

```
1. def hi(name="sir"):
2.         print("Hello " + name + "! How are you today?")
```

In this case you can call the function with and without the argument. Calling:

```
1. hi()
2. hi("James")
```

```
Hello sir! How are you today?
Hello James! How are you today?
```

```
1. def foo(bar, zoo="blah"):
2.     ...
```

is valid. Meanwhile:

```
1. def foo(bar="blah", zoo):
2.     ...
```

is not.

Return values

Lets have a look at one of the code snippets that was covered previously in the exercises:

```
1. N = 10
2. distance_in_kilometers = 100 * N
3. distance_in_meters = distance_in_kilometers * 1000
```

What if you want to encapsulate this logic in a function in such a manner that you could pass time in hours and speed in km/h and somehow get back a distance traveled in meters?

This logic could be implemented using the following function:

```
1. def get_distance(speed, duration):
2.     distance_in_kilometers = speed * duration
3.     distance_in_meters = distance_in_kilometers * 1000
4.     return distance_in_meters
```

The function's last line contains a return statement. The best way to understand what it does is to have a look at several invocations of the function:

```
1. speed = 100
2. initial_distance = get_distance(speed, 0)
3. distance_after_a_day = get_distance(speed, 24)        # day is 24 hours
4. distance_after_a_week = get_distance(speed, 24 * 7)     # week is 7 days
```

As you may have guessed, the return statement allows you to store the output of the function in a variable. After executing the lines above the variables shall have the following values:

variable	value
initial_distance	0
distance_after_a_day	2400000
distance_after_a_week	16800000

Exercises

PROBLEM: There is a mathematical function f(x,y) = x^y + 100.
Implement the function in Python.

ANSWER:

```
1. def function(x, y):
2.     return x ** y + 100
```

Exercises

PROBLEM: Which function definitions are valid?

```
1. @foo(bar):
2. _foo(bar = 1):
3. foo(bar = 2, bla = 1):
4. foo(bar = 2, bla):
5. foo($bar, bla = 1):
```

ANSWER:

1. invalid: symbol @ in function's name
2. valid
3. valid: there are several arguments with default values
4. invalid: argument without default value
5. invalid: symbol $ in argument's name

Exercises

PROBLEM: Change the function below to return the greeting instead of printing it:

```
def hi():
    greeting = "Hello world!!!"
    print(greeting)
```

ANSWER:

```
def hi():
    greeting = "Hello world!!!"
    return greeting
```

If/Else and conditional statements

There are a lot of uses for the simple statements and operations we have covered so far in this book. However, the main purpose of any programming is to implement some kind of logic.

If/else blocks are a core syntax element in Python. They let you conditionally control the flow of the program.

In the following scenario, you want to know if your car is travelling too quickly. If you are exceeding 100 km/h, the function will print "Too fast!" If you drive below the speed limit, the function will print "You are driving below the speed limit. Well done."

This is what that code would look like.:

```
1. def drive(car_speed):
2.
3.     if car_speed > 100:
4.         print("Too fast!")
5.     else:
6.         print("You are driving below the speed limit. Well done")
```

```
>>> drive(70)
You are driving below the speed limit. Well done
>>> drive(120)
Too fast!
```

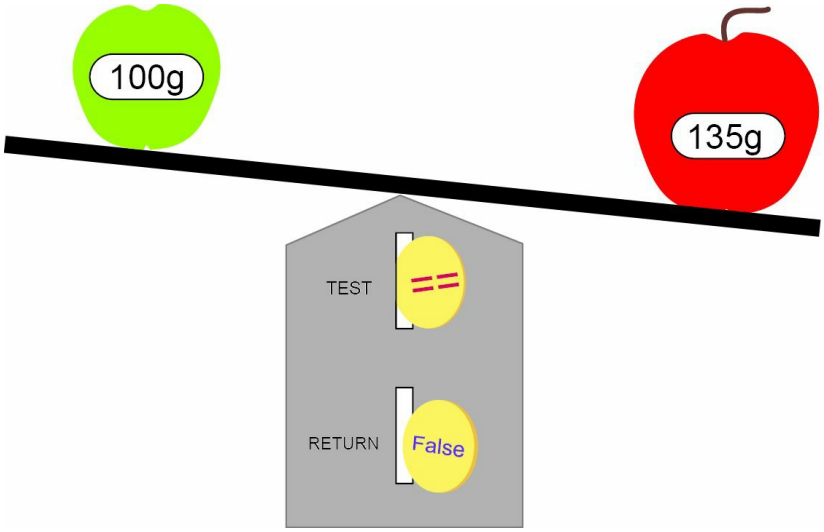
There are multiple things to note in the code above.

First, the > operator. It's a Boolean comparison operator, the same as the one used in maths. It returns True if the first item is larger than the second, and False otherwise.

Second, the if/else syntax. If is_too_fast is True, the first block of code is executed. If not, the second block of code is executed.

Third, the indentation. The statements inside the function are indented, and then the statements inside the if/else block are further indented.

Boolean expressions




```
greenApple = 100
redApple = 135

print(greenApple == redApple)       #test if 100 and 135 are equal
#False

print(greenApple < redApple)       #test if 100 is less than 135
#True

print(greenApple != redApple)       #test if not equal
#True
```

```
>>> bool(1)
True
>>> bool("")
False
>>> bool(None)
False
>>> bool("blah")
True
>>> bool(0)
False
>>> bool(True)
True
```

In Python any value can be interpreted as True or False. Among the data types values 0, "" - an empty string, None and False itself are False. Any other value is True.

You may check if the value is True or False by converting the value to a boolean type:

Also there are several built-in comparison operators like > return True or False when called.The table below provides an overview of these operators.

Operator	Statement	Statement's return value
Less	`1 < 2` `4 < 3`	`True` `False`
Less or equal	`2 <= 2`	`True`

	`4 <= 3`	`False`
Greater	`2 > 1` `3 > 3`	`True` `False`
Greater or equal	`2 >= 2` `3 >= 4`	`True` `False`
Equal	`"foo" == "foo"` `0 == 1`	`True` `False`
Not equal	`"foo" != "bar"` `11 != (10 + 1)`	`True` `False`
And (both parts must be True)	`3 and "foo"` `None and True`	`True` `False`
Or (one of parts must be True)	`1 or None` `None or False`	`True` `False`
Not (opposite to the target expression)	`not None` `not True`	`True` `False`

Note, comparison operators have higher precedence than the logical operators "and", "or" and "not". All maths operators have higher precedence than comparison operators. Here are several compound examples to illustrate how to use boolean operators:

```
>>> 2 * 6 + 1 < 2 * (6 + 1)
True
>>> 2 * 6 + 1 == 2 * (6 + 1) - 1
True
>>> 2 ** 3 == 8 and 2 ** 2 == 16
False
>>> (2 ** 2 == 4 and False) or (not (16 < 15 or 1 == 3))
True
```

Alternative branches

In many cases the logic can be more sophisticated than just two conditions.

Let's modify the speed function we created before to respond to speeds over 200 km/h by printing "You are insane, man!!!" and printing "The optimal speed for your car." when the speed is between 70 and 80 km/h.

```
1. def drive(car_speed):
2.     if car_speed > 200:
3.         print("You are insane, man!!!")
4.     elif car_speed > 100:
5.         print("Too fast!")
6.     elif car_speed > 70 and car_speed < 80:
7.         print("The optimal speed for your car.")
8.     else:
9.         print("You are driving below the speed limit. Well done")
```

You might notice that in the code above we don't use a separate Boolean variable. Instead, we place the comparison operator inline with the if statement. The "and" statement links two comparison operators together.

We also use the keyword "elif". This defines an alternate branch. Be careful with the order of branches! If more than one branch is True, only the first one will be used. For example:

```
1. a = 10
2. if a > 1:
3.     print("Big")
4. elif a > 5:
5.     print("Really big")
6. elif a == 10:
7.     print("Approaching enormity")
8. elif a > 10:
9.     print("Enormous!")
```

This code will only ever print "Big", and the other statements will be ignored.

Quiz

Try to answer these questions without using the interpreter.

1. What are the boolean values - bool(a) - for the following variables:

```
1. a = False
2. a = "0"
3. a = 0 * 18
4. a = "None"
```

2. What is the output of the following statement:
   (2 ** 2 == 4 and False) or (not (16 > 15 or 1 == 3))
3. Name Python's logical operators
4. What will be printed as a result of the following code?

```
1. x = 5
2. if x == 25:
3.     print ("Quarter")
4. elif x == 10:
5.     print ("Dime")
6. elif x == 5
7.     print ("Nickel")
8. elif x == 1:
9.     print ("Penny")
10. else:
11.     print ("Not a coin")
```

5. Suppose x=5, y=0, and z=1. Are the following expressions True or False?

```
1. x == 0
2. x >= y
3. 2 + 3 == x and y < 1
4. not (z == 0 or y == 0)
5. x = 3
```

6. What does the following code print?

```
1. x = 0
2. if x != 0:
3.     print("x is nonzero")
```

ANSWER:

1. a) False
   b) True - this is a string with a character 0 not an integer
   c) False - 0 * 18 is 0
   d) True - again “None” is a string not the actual None
2. False - the expression inside the “not” statement is True, making that statement False, and the first statement is also False.
3. and, or, not
4. Nickel. x == 25 and x == 10 are False, so the first True statement is x == 5.
5. a) False. The operator == tests for equality, and 5 does not equal 0.
   b) True. 5 is greater than or equal to 0.
   c) True. Both 2 + 3 == x and y < 1 are True.
   d) False. y == 0 is True, so z == 0 or y == 0 is True, and its negation is False.
   c) True. The = operator assigns the value 3 to x, and the value 3 evaluates to True.
6. Nothing. The expression is False, so the code is not executed.

Exercises

PROBLEM: Write an expression that tests if a variable x is less than or equal to 100.

ANSWER:

```
1. x <= 100
```

Exercises

PROBLEM: Write an expression that tests if a variable x is between 3 and 10, inclusive.

ANSWER:

```
x >= 3 and x <= 10
```

Exercises

PROBLEM: Write an expression that tests if a variable *x* is even or a multiple of 5.

ANSWER:

```
1. (x % 2 == 0) or (x % 5 == 0)
```

Exercises

PROBLEM:

There is a function with the following declaration: person_is_old(age) which returns True or False based on the incoming age as an integer.
Write a function that uses the one mentioned above so that it prints “Go to work!” if the person is not old and “You deserved to have a long vacation” otherwise. Your function must also accept the age as its argument.

ANSWER:

```
1. def tell_what_to_do(age):
2.     if person_is_old(age):
3.         print(“You deserved to have a long vacation”)
4.     else:
5.         print(“Go to work!”)
```

Classes and objects

One of the key principles of object-oriented programming is encapsulation - hiding the details of the inner workings of the system, so that the user doesn't need to know how it works in order to use it. The part of the program that the user interacts with is called the "interface".

Just as the driver of a car needs tools like a steering wheel and a gas pedal to use the car's capabilities, users of your code need to use syntax constructs called methods.

Here, we have implemented a car in Python:

```
1.  class Car:
2.      def turn_left(self):
3.          pass
4.      def turn_right(self):
5.          pass
6.      def accelerate(self):
7.          pass
8.      def slow_down(self):
9.          pass
10.     def open_a_window(self):
11.         pass
```

In the code above, we have created a class. A class is an entity that has specific behaviours. These behaviours take the form of methods, which are the same as the functions we have used previously, and are indented in the same way. The methods of a class are given the class ("self") as their first parameter. The "pass" keyword tells Python that this code block does nothing.

There are a variety of different cars that this class could represent - BMW, Mercedes, Audi, Fiat, Tesla, and many others. Let's create a car with some parameters.

```
1.  bmw_x6 = Car(model = "BMW X6", max_speed = 230)
```

In the code above you can see that the car object was created with two parameters: model and max_speed. The original car code has nothing that would support them. In order to implement this support you must introduce an __init__ method:

```
1.  class Car:
2.      def __init__(self, model, max_speed):
3.          self.model = model
4.          self.max_speed = max_speed
5.          self.speed = 0
6.      ...
```

The code above takes your new car object ("self") and uses the dot notation to assign the car's model and max_speed. The car begins with a speed of zero.

The __init__ method shouldn't contain a return statement - it's just used to build your object. This method is called the "class constructor".

Accelerating and slowing down should decrease and increase the car's speed appropriately:

```
1. class Car:
2.   ...
3.   def accelerate(self, speed_difference):
4.     self.speed += abs(speed_difference)
5.     self.speed = min(self.speed, self.max_speed)
6.   def slow_down(self, speed_difference):
7.     self.speed -= abs(speed_difference)
8.     self.speed = max(self.speed, -5)
9.   ....
```

The internal state of the object is changed using the dot notation.

The min and max functions are used to guarantee that the car never exceeds the maximum speed, or travels below -5 km/h (reverse transmission)

The following code would accelerate the car by 30 km/h.

```
1. bmw_x6.accelerate(30)
```

When you're creating more than one class, you may want some of them to have relationship with each other. For example, a Car is a Vehicle, and a Bus is also a Vehicle. Although they share many similarities, they have very different internal structures.

To reflect similarities and differences in objects, Python has a feature called inheritance.

In this example, we implement a Vehicle class, which has model and max_speed parameters like the Car class.

```
1. class Vehicle:
2.   def __init__(self, model, max_speed):
3.     self.model = model
4.     self.max_speed = max_speed
5.     self.speed = 0
6.   def accelerate(self, speed_difference):
7.     self.speed += abs(speed_difference)
8.     self.speed = min(self.speed, self.max_speed)
9.   def slow_down(self, speed_difference):
10.    self.speed -= abs(speed_difference)
11.    self.speed = max(self.speed, -5)
12.  ...
```

Now we want to create Car and Bus classes; the Car class is identical, but the Bus class

doesn’t have a reverse transmission.

```
class Car(Vehicle):
    pass
class Bus(Vehicle):
    def slow_down(self, speed_difference):
        super().slow_down(speed_difference)
        self.speed = max(self.speed, 0)
```

The word “Vehicle” between the perentheses after the class name makes the Car and Bus classes inherit from Vehicle.

In the Bus class, the slow_down method overrides the behavior of the parent class. The super() statement makes a call to the parent class to find the new speed, and then makes sure that this speed is not below 0 km/h.

Quiz

PROBLEM:

1. What would a software engineer call a collection of behaviors that the user can use?
2. What is the name of the relationship between classes?
3. What is the general term for methods used to create objects in Python? For example, the following method: __init__(name, max_speed).
4. What is missing from the method in the previous question?

ANSWER:

1. Interface
2. Inheritance
3. class constructor.
4. self argument is missing

Exercises

PROBLEM: Write Python code that would make BMW X6 accelerate by 100 km/h

ANSWER:

```
1. bmw_x6.accelerate(100)
```

Exercises

PROBLEM: Introduce a Bike class that is also a Vehicle. The main feature of a bike is that it can never ever be faster than 30 km/h.

ANSWER: You just need to make sure that max speed is capped by 30 km/h in Bike's constructor

```
1. class Bike(Vehicle):
2.     def __init__(self, name, max_speed):
3.         max_speed = min(max_speed, 30)
4.         super().__init__(name, max_speed)
```

Exercises

PROBLEM: Add to your code from the previous exercise a new method called show_status that prints something like “The bike is BMX, its speed is 27 km/h”

ANSWER: Method’s implementation is supposed to look the following way:

```
1. def show_status(self):
2.     print(“The bike is “ + self.name + “, its speed is “
3. str(self.speed) + “ km/h”)
```

Collections

In many cases, you want to perform the same action on a lot of very similar objects. For example, in real life, you might want to iron your socks. It wouldn't make sense for you to consider each one individually as a unique object requiring special treatment.

In programming, this is where Python collections come in handy. A good analogy for collections is a set of lockers.

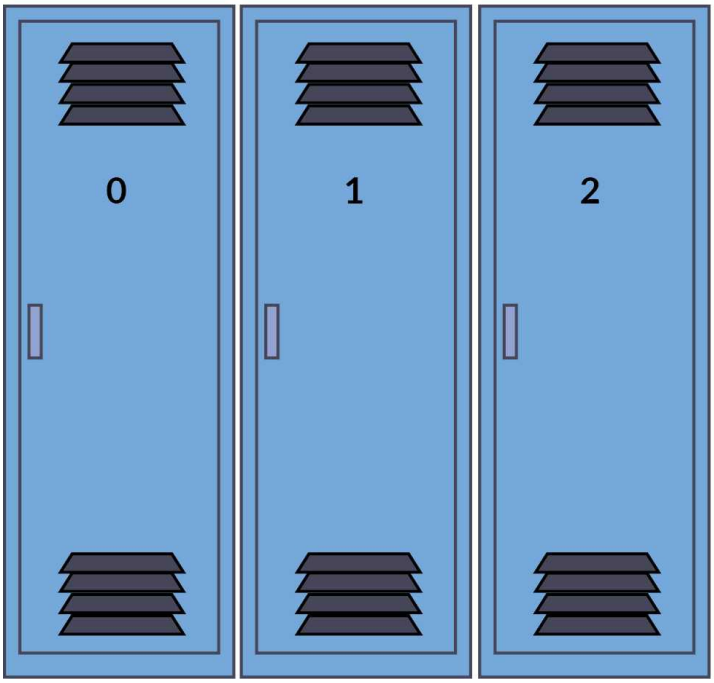
Python has two types of lockers: ones with name tags, and anonymous ones, indexed only by number.

To access items in the anonymous locker room, you can either go through all the lockers one by one, or access a specific one by an integer index. In Python, these lockers are represented by lists and tuples.

For lockers with name tags, you can either go through each locker one by one, or you can look at a specific name tag. In Python, these lockers are represented by dictionaries.

This chapter provides an overview of built-in Python collections.

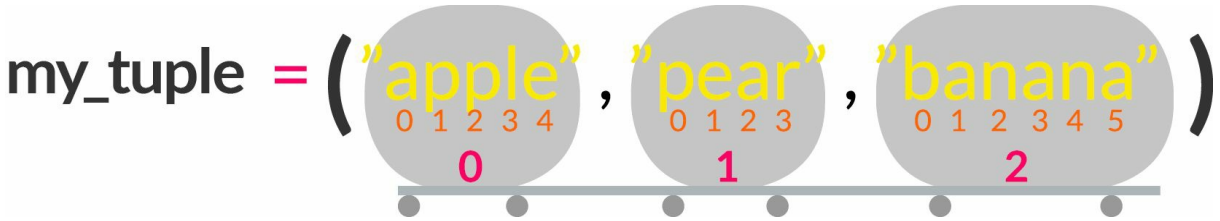
Lists and tuples use integers for indexing.



Dictionaries use keys for indexing.

Tuples

Tuples are immutable arrays of variables.



You can access the elements of tuples in the following way:

```
>>> my_tuple = ("apple", "pear", "banana")
>>> my_tuple[1:]          # from second to the end
'pear', 'banana'
>>> my_tuple[2][2]        # 3rd character of 3rd item
'n'
```

"Immutable" means that once you define a tuple, you can't change it.

```
>>> my_tuple[2] = "mango"        # replace 'banana' with 'mango'
TypeError: 'tuple' object does not support item assignment
['apple', 'pear', 'mango']
>>> my_tuple[2][0] = "B"         # try to change the 'b' with "B" and you
                                 # will get an error
TypeError: 'str' object does not support item assignment
>>> new_tuple = ('foo', 'bar', 'zoo', 'loo')
>>> type(new_tuple)
<class 'tuple'>
>>> new_tuple[0]
'foo'
>>> new_tuple[3]
'loo'
>>> new_tuple[1:3]
('bar', 'zoo')
>>> len(new_tuple)
4
>>> new_tuple * 2
('foo', 'bar', 'zoo', 'loo', 'foo', 'bar', 'zoo', 'loo')
>>> new_tuple + ('blah', 'blah')
('foo', 'bar', 'zoo', 'loo', 'blah', 'blah')
```

You can see from the code snippet above that tuples behave similarly to Python strings - however, strings are arrays of just characters, whereas tuples can contain any kind of variable.

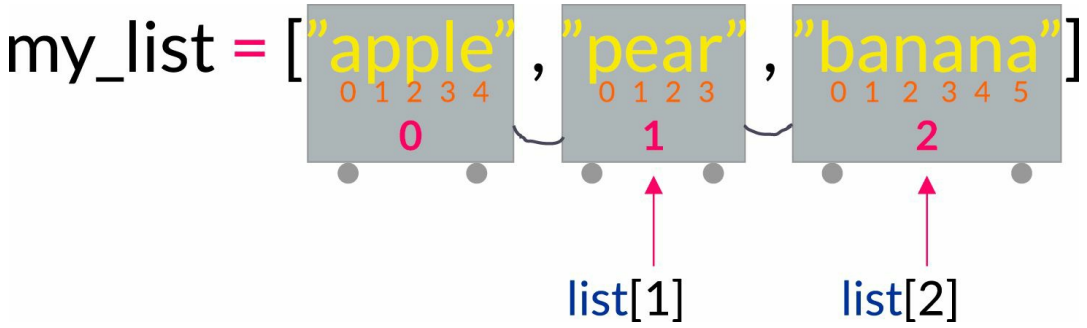
You can slice, index, concatenate, and multiply tuples in the same way that you can with strings.

Syntactically, tuple items must be surrounded by parentheses, and separated by commas. Even tuples containing a single item must have a comma: (1,) is a tuple, but (1) is not.

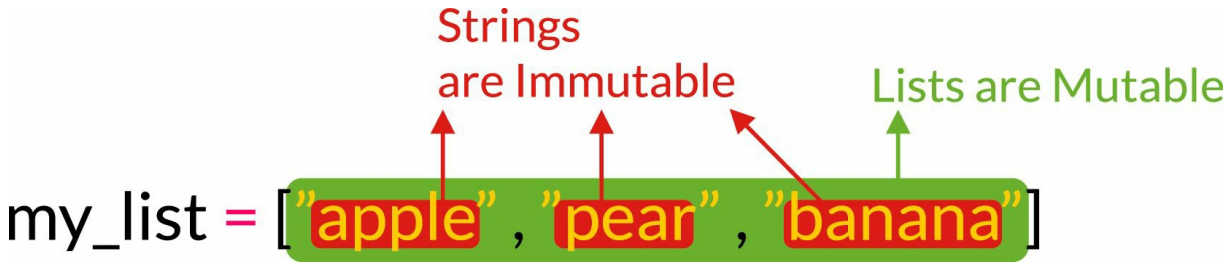
Lists

Lists are like tuples that you can change after you've created them.

Lets define a simple list with three items:




```
>>> my_list[1:]          # from second to the end
'pear', 'banana'
>>> my_list[2][2]        # 3rd character of 3rd item
'n'
```




```
>>> my_list[2] = "mango"          #replace 'banana' with 'mango'
>>> my_list
['apple', 'pear', 'mango']
>>> my_list[2][0] = "B"          #try to change the 'b' with "B" and you
                                 #will get an error
TypeError: 'str' object does not support item assignment
>>> new_list = ['foo', 'bar', 'zoo', 'loo']
```

As you may have noticed, list items are supposed to be enclosed by square brackets.

Now lets remove the 3rd item and change the first one to "blah":

```
>>> new_list.pop(2)
>>> new_list[0] = "blah"
>>> new_list
['blah', 'bar', 'loo']
```

You can add items to the end of the list:

```
>>> new_list.append("new_item")
```

And pop the items from the end of the list:

```
>>> new_list.pop()
'new_item'
```

You can also add the item to a specific position in the list:

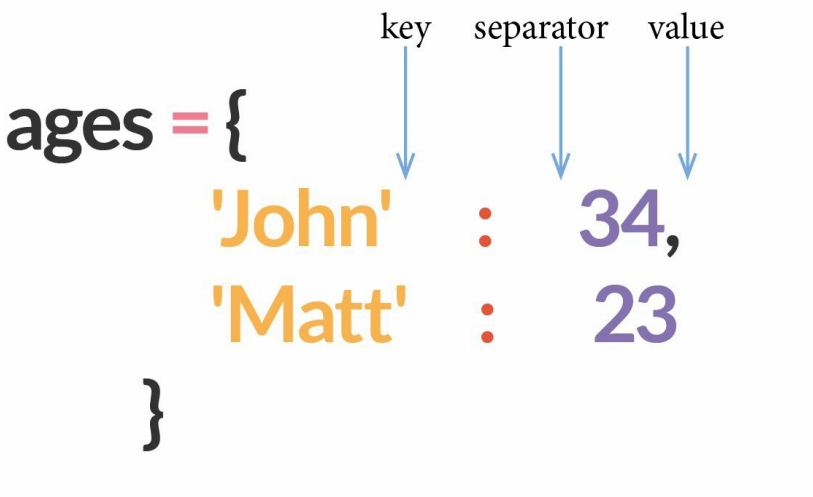
```
>>> new_list = ['foo', 'bar', 'zoo', 'loo']
>>> new_list.insert(1, 'John')
>>> new_list
['foo', 'John', 'bar', 'zoo', 'loo']
```

When you add an item to a list, it takes the location you specify and shifts all the following items to the right.

Dictionaries

A dictionary is a collection of keys and values, which works much the same as a list or a tuple. Instead of an integer index, dictionary values can be addressed by indices of almost any type.

Lets create a dictionary with people names as keys and their ages as values:




```
ages = {
   "John": 34,
   "Matt": 23,
   "Natasha": 27,
   "Gabriella": 33
    }
```

The key-value pairs must be enclosed in curly brackets and separated by commas. Keys and values must be separated by a colon.

The keys in the dictionary must be unique, but the values can be the same. A key can be almost any type of variable.

The following is also a valid dictionary:

```
class SomeClass:
   pass
obj1 = SomeClass()
obj2 = SomeClass()
a_dict = {
   obj1: 11,
   obj2: 23
    }
```

You may fetch John's age the following way:

```
johns_age = ages["John"]
```

In the same way, you can set the value of elements by key:

```
1. ages[“John”] = 999
```

The above statement either changes the value of an existing key or creates a new key-value pair if none exists.

You can also remove a key-value pair from the dictionary.

```
1. ages.pop(“John”)
```

Quiz

PROBLEM:

1. Which collection resembles a locker room with name tags on the lockers? Which one resembles an anonymous locker?
2. What is the main difference between lists and tuples?
3. Can a float be used as a key in a dictionary?
4. How many keys called "John" can you have in a dictionary?
5. Is this a tuple: (123)?

ANSWER:

1. dictionary/list and tuple
2. lists can be modified, tuples can't
3. yes, why not?
4. exactly one - all keys must be unique
5. no - a comma was forgotten

Exercises

PROBLEM: Using Python get Natasha's age.

ANSWER:

```
ages['Natasha']
```

Exercises

PROBLEM: Get rid of Matt.

ANSWER:

```
ages.pop('Matt')
```

Loops

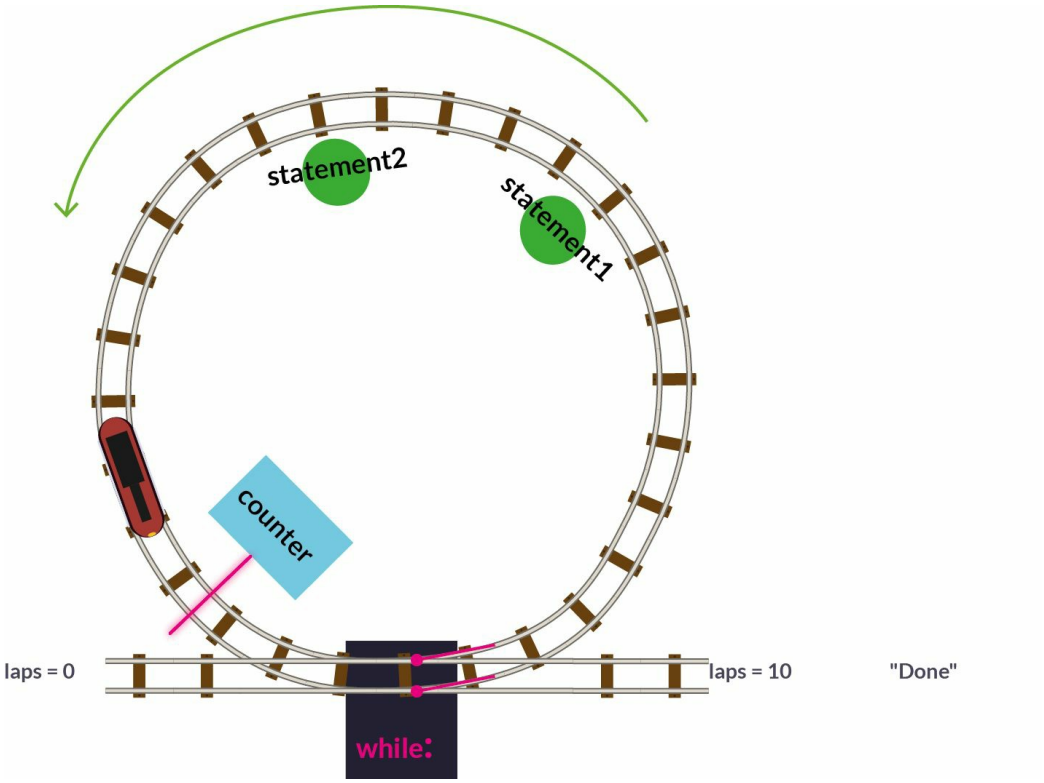
Collections can be used to store data in, but you can also go through them and do something to each item individually.

In Python there are two types of loop that you can use: while and for.

A while loop carries on repeating as long as a certain condition is true.

The statements inside a while loop must be indented, like any other code block.
The loop must be defined using the "while" keyword, a Boolean statement, and a colon.

The Boolean statement could be a function call, a variable, or a comparison statement.

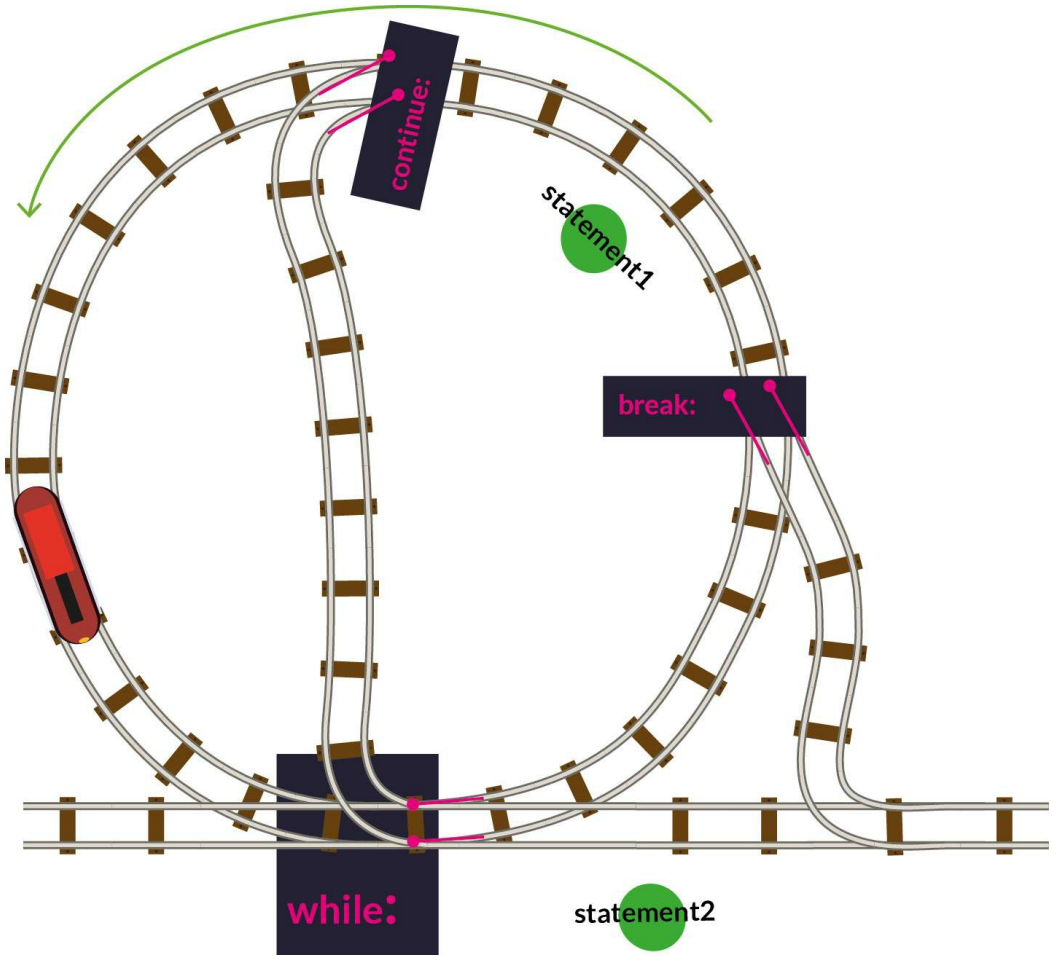


This code will print numbers 1 to 10

```
1. laps = 0
2. while laps < 10:     #test
3.     laps += 1        #counter
4.     print (laps)     #statement1
5.     print ("and")   #statement2
6. print ("Done")      #loop exit
```

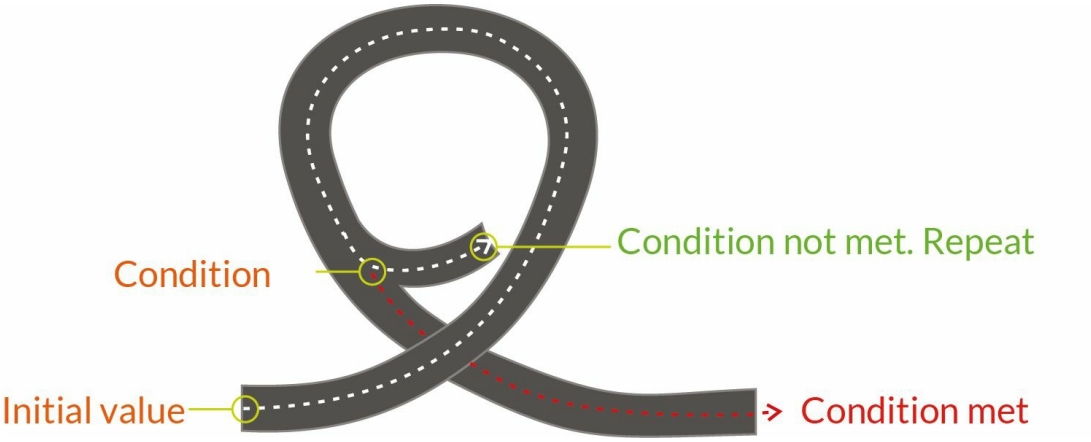
While

Python provides break and continue statements to have even better control on your loop.




```
1. while <test1>:
2.     <statement1>
3.     if  <test2>: break         #loop exit, skip else
4.     if  <test3>: continue        #go to the top <test1>
5. else:
6.     <statement2>           #run if break didn't happen
```

Loops



You can use a "for" loop to iterate though a collection.
Let's say you have the following list:

```
1. a = ["one", "two", "three"]
```

To print each string individually, you could use this loop:

```
1. for item in a:
2.     print(item)
```

The loop must be defined using the "for" keyword, the name of the new variable to hold each individual item, the "in" keyword, the name of the collection, and a colon. The statements inside a "for" loop must also be indented.

You can escape from a loop using the "break" command:

```
1. numbers = [1, 2, 5, 6, 7, 8, 9]
2. for number in numbers:
3.     print("Looking at: " + str(number))
4.     if number > 6:
5.         print("Too big: " + str(number) + "!")
6.         break
```

The above code gives the following output:

The iteration stops after number 7, meaning that numbers 8 and 9 are not included.

You can also skip particular items in a collection using the "continue" keyword:

```
numbers = [1, 6, 7, 8, 9, 2, 5]
for number in numbers:
    if number > 6:
      continue
    print("Looking at: " + str(number))
```

The above code gives the following output:

Here, we ignored numbers 7, 8, and 9.
Dictionaries can be iterated through in the same way.

```
ages = {
    "John": 34,
    "Matt": 23,
    "Natasha": 27,
    "Gabriella": 33
     }
for name in ages:
    print(name)
```

Quiz

PROBLEM:

1. If you wanted to write code that triggered an alarm every ten minutes, as long as you were sleeping, should you use a “for” or a “while” loop?
2. How do you completely stop a while loop? Give two options.
3. How do you skip a specific item in a for loop? Give two options.

ANSWER:

1. while
2. Change while loop’s condition to evaluate to False or use a break keyword.
3. Isolate the code for item processing within an if statement or use a continue keyword.

Exercises

PROBLEM: Write some code to print out the name of the oldest person inside the ages dictionary. This will use knowledge from earlier chapters in the book.

ANSWER:

```
ages = {
   "John": 34,
   "Matt": 23,
   "Natasha": 27,
   "Gabriella": 33
    }
oldest_person = None
current_biggest_age = 0
for name in ages:
   age = ages[name]
   if age > current_biggest_age:
      oldest_person = name
      current_biggest_age = age
print(oldest_person)
```

File operations

Most file operations boil down to: opening files, reading files, writing data into files, and closing files. Usually the data you write into a file will be textual.

Let's say you want to write a list of integers into a file so that every integer is on a separate line. First, open the file:

```
1. the_file = open("integers.txt", "w")
```

The first argument to the "open" function is the path to the file, and the second argument tells Python that you want to write data into the file. This means that the file doesn't have to exist before you begin.

The function returns a file object that you can use later on.

First let's define an array of integers and write them to the file as strings.

```
1. integers = [1, 2, 3, 4, 5]
2. for integer in integers:
3.     the_file.write(str(integer) + '\n')
```

The "write" method takes in a string as an argument and writes it to the file.

Note the "\n" symbol - this is the newline character. It starts a new line in the file.

When you're finished with the file, you have to close it:

```
1. the_file.close()
```

You can also read data from a file.

First, open the file without using the "w" flag:

```
1. the_file = open("integers.txt")
```

Get all the lines from the file:

```
1. lines = the_file.readlines()
```

The readlines() functions returns all of the lines in the file as a list of strings.

Let's print out the lines:

```
1. for line in lines:
2.     print(line)
```

After you execute the code above you shall see the following on the screen:

```
1
2
3
4
5
```

And as with writing to a file - close it when you are done:

```
1. the_file.close()
```

Quiz

PROBLEM:

1. What does a newline character look like?
2. What are you supposed to do with the file once you are done working with it?
3. Which method is used to put data into the file?
4. What method should you use to get all the lines from the file?

ANSWER:

1. \n
2. Close it
3. write
4. readlines

Exceptions

Sometimes something unexpected happens that stops a program from executing as it should. These events are called “exceptions”.

```
>>> 1 / 0
```

After you hit enter, you’ll see the following output:

```
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
ZeroDivisionError: division by zero
```

This is called an exception traceback. It shows you which functions or methods were called before the exception happened. The last line tells you which exception was raised and gives you some additional, human-readable information.
In Python you can achieve this via a try/except block.

What if an exception happens in the middle of something important?

Lets print “division by zero. But it is totally fine.” instead of throwing a traceback.

```
1. try:
2.     1 / 0
3. except ZeroDivisionError as e:
4.     print(str(e) + “. But it is totally fine.”)
```

In the code above you can see that the statements you want to make an attempt to execute go into the try block. And all the error handling code goes under except.

except keyword must be accompanied with a class of the exception. It also can be followed by an as keyword and a variable which is supposed to store a human readable description of the error.

Custom exceptions

You can define your own exception classes to further on notify that something extraordinary happend inside your code:

```
1. class YourException(Exception):
2.     pass
```

All custom exceptions must have a built-in Python exception as a superclass. The exceptions do not have to implement any other logic.

In order to throw an exception you should use the raise keyword:

```
1. raise YourException("Human-readable message")
```

Once you execute the code above, you shall see a traceback with "Human-readable message" string as an extra detail.

Handling custom exceptions is not different from handling the built-in ones:

```
1. try:
2.     raise YourException("Human-readable message")
3. except YourException as e:
4.     print("Your exception just occured with a message: " + str(e))
```

Quiz

PROBLEM:

1. Which block is supposed to be used to handle exceptions?
2. What is the name of the long text blob that is printed once an exception takes place?

ANSWER:

1. try/except block
2. traceback

Part II: Standard Library

Import statement and Python modules and packages

All source code in Python, as you may have noticed, is stored in various files called modules. Each module has a .py extension. Different modules are logically combined together into collections called packages. A package is a plain folder with __init__.py file. E.g. the following is a package:

```
foobar/
  __init__.py
  one.py
  two.py
  three.py
```

And this one is not:

```
foobar/
  one.py
  two.py
  three.py
```

Default Python installation comes with a significant amount of packages.
This set of packages is called a standard library.

Import statement

If you want to use functionality stored in another package you must use an import statement.

Lets print your user name on the screen using Python interpreter:

```
>>> import getpass
>>> getpass.getuser()
'gunther'
```

The snippet above shows how to import the most top level package. As you can see members of the package are supposed to be accessed via the dot operator.

You may avoid using the dot operator by importing the thing you need directly:

```
>>> from getpass import getuser
>>> getuser()
'gunther'
```

To indicate the source of the import you must use from keyword.

Sometimes you may want to use multiple entities from the same package. In this case it is simpler to import all the entities straight away:

```
>>> from sys import pydebug, flags
```

Note, the entities kept inside Python packages can be virtually any Python constructs. A class, a function, a variable, a module or even a package (nested packages are possible).

Quiz

PROBLEM:

1. What is the synonym of Python source file?
2. What is the term for folders with Python modules?
3. What is the name of a collection of Packages that comes together with default Python installation?
4. How to import module path from os package?

ANSWER:

1. module
2. package
3. standard library
4. from os import path

File operations using os module

Even though it is possible to work with files using builtin Python functions and methods they do not cover all the scenarios that you may face when manipulating computer file system.

This chapter provides an overview of all the common tasks you may face on the lower level when processing the data and Python means of managing with these tasks.

A quick note: some of the examples below use Linux's path structure. To play with them on Windows replace a slash (/) with a backslash (\).

To begin with lets say you have got the following folder structure (e.g. created using a plain file manager):

```
/parent/
   foo/
      one.txt
      two.txt
      three.txt
```

First navigate to the parent directory of foo using a plain terminal command:

```
cd /parent
```

Lets verify that directory's path is what we expect it to be.

To get the absolute path of the current directory you ought to employ abspath function and curdir property of os.path module:

```
>>> from os.path import abspath, curdir
>>> curdir
'.'
>>> abspath(curdir)
'/parent'
```

curdir variable holds a string pointing to the current directory - .. abspath expects any path as its argument and returns the absolute path.

Now remove one.txt and rename two.txt into bar.txt. To do this you will need two functions from the standard library's os module: remove and rename respectively.

File operations using os module

```
>>> from os import remove, rename
>>> remove("foo/one.txt")
>>> rename("foo/two.txt", "foo/bar.txt")
```

As you may have noticed, in order to get rid of the file you must pass its name to the function and in order to rename the file apart from supplying the original name you are also supposed to pass the string that will be assigned as the file's new name.

Note, when supplying the name of the file you must give either a relative or an absolute path to the file. In case of the absolute path everything is plain and simple - it should start with the root of your file system.

The absolute path could look the following way on UNIX and Windows respectively:

```
/Blah/Blah/foo/one.txt
C:\Blah\Blah\foo\one.txt
```

Relative paths are calculated with your current directory.
Lets say you are in a directory /foo/bar/one and you want to move to /foo/blah/two.
In this case the relative path shall be: ../../two. Two dots represent a parent directory.
And since you need two navigate two levels above - a sequence of ../../ is required.

Due to the fact that different delimiters are used on *nix and Windows the remove/rename code snippet above has a serious flaw. It is not multi-platform.
In order to transform it into one you should use join function from os.path module:

```
>>> from os import remove, rename
>>> from os.path import join
>>> remove(join("foo", "one.txt"))
>>> rename(join("foo", "two.txt"), join("foo", "bar.txt"))
```

You pass all the parts of the path to the join function as strings, and then the operating system-specific path delimiter is used to produce a string.

You can list all the files and folders in a particular directory:

```
>>> from os import listdir
>>> listdir("foo")
["bar.txt", "three.txt"]
```

Now let's try to remove the foo directory.

```
>>> from os import rmdir
>>> rmdir("foo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 39] Directory not empty: 'foo'
```

Oops! We've cause an exception. The directory can't be deleted as long as it contains anything inside.

Quiz

PROBLEM:

1. What major difference between Windows and Linux filesystems make it more difficult to write cross-platform Python programs?
2. What is the value of os.path.curdir?
3. Which string represents the parent of the current directory?

ANSWER:

1. Different path delimiters: / on Unix and \ on Windows
2. .
3. . .

Exercises

PROBLEM:

Fix the "rmdir" code from the last example so that it works.

ANSWER:

To do so you just need to get rid of all the files inside first:

```
>>> remove("foo/bar.txt")      # you renamed it
>>> remove("foo/three.txt")
>>> rmdir("foo")
```

Exercises

PROBLEM: Make the following function cross-platform:

```
1. def show_foobar_here(path):
2.     print(path + "/foobar")
```

ANSWER:

```
1. from os.path import join
2. def show_foobar_here(path):
3.     print(join(path, "foobar"))
```

Exercises

PROBLEM:

Using the isdir function from the os.path module, find the folder inside your current directory.

Iterate through the files and folders, and then throw a custom DirFound exception with the absolute path of the directory as the human-readable description.

ANSWER:

The following would be a sufficient solution:

```
1. from os.path import isdir, curdir, abspath
2. from os import listdir
3. class DirFound(Exception):
4.   pass
5. def detect_dirs():
6.   for item in listdir(curdir):
7.     if isdir(item):
8.       raise DirFound(abspath(item))
```

Regular Expressions

Although you can use the "in" operator to check if a substring is present within a string, it doesn't tell you where the substring is located, nor does it let you match the substring non-precisely.

Take this string:

```
>>> lorem_text = "Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut`labore et dolore magna aliqua."
```

You want to find out if this text contains something like "dolor BLA BLA BLA amet" - the words "dolor" and "amet" in that order, with any amount of text in between.

To do this, you can use the re.search method:

```
>>> import re
>>> found = re.search("dolor .*amet", lorem_text)
>>> found
<_sre.SRE_Match object; span=(12, 26), match='dolor sit amet'>
```

The "search" function takes a regular expression pattern as its first argument, and the text to be searched as its second argument.

The regular expression patter is a flexible description of what you're looking for in the text. The format is covered later in this chapter.

Look at the return value of the search function: it's called a MatchObject, and it tells you where the substring was found in the original text.

If the pattern can't be found, then the function returns None:

```
>>> match = re.search("found", lorem_text)
>>> type(match)
<class 'NoneType'>
```

You may get start and end indices of the substring from the match object:

```
>>> found.start()
12
>>> found.end()
26
```

and access the substring matched by the pattern:

```
>>> found.group()
'dolor sit amet'
>>> lorem_text[12:26]
'dolor sit amet'
```

Note: if multiple occurrences of the patter are found, only the first one is returned. If you need to go through all of the occurrences, use the re.finditer function:

```
>>> list(re.finditer("dolor .*amet", lorem_text))
[<_sre.SRE_Match object; span=(12, 26), match='dolor sit amet'>]
```

If you want to see if a string matches a certain pattern, you should use the re.match

function, which tries to apply the pattern to the whole string:

```
>>> bool(re.match("[0-9][a-z]*", "0dgfgfg"))
True
>>> bool(re.match("[0-9][a-z]*", "dgfgfg"))
False
```

The symbols ".*" in the regular expression pattern match to zero or more arbitrary characters. A dot represents an arbitrary character, and a star represents a cardinality. This is the basic structure of any regular expression pattern: a symbol type, followed by its cardinality.

The table below provides an overview of the most frequently-used cardinalities.

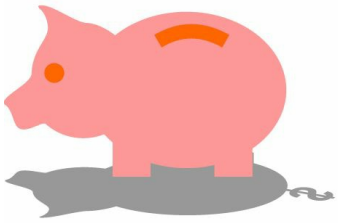
Cardinality	Description
*	Zero or more
+	One or more
?	Zero or one
{m}	Exact number of symbols
{m, n}	At least m and n at most

This is how to easy remember some of them:

Matches the beginning of a line

Also can mean "NOT"

```
>>>
>>>
>>>                                    # not line
```



Matches the end of a line

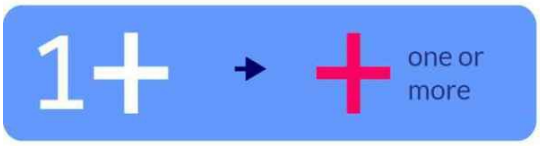
```
>>>
>>>
```

re

= "The quick brown fox jumps over the lazy dog quick"

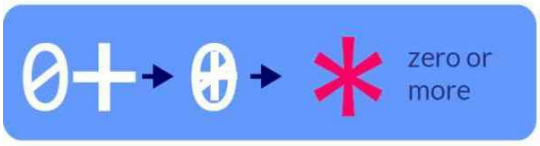
or

```
x = re.findall("quick|dog", txt)
['quick', 'dog']
```



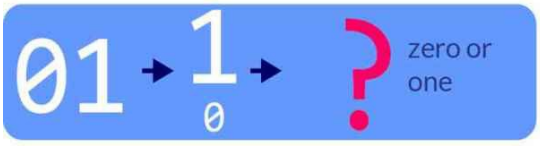
one or more

```
x = re.findall("quick+", txt)
['quick', 'quick']
```



zero or more

```
x = re.findall("quick*", txt)
['quick', 'quick']
```



zero or one

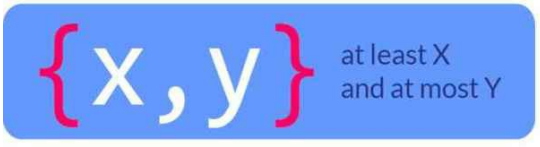
```
x = re.findall("quick?", txt)
['quick', 'quick']
```

group

```
x = re.findall("quick ([a-z]+)", txt)
['brown']
```

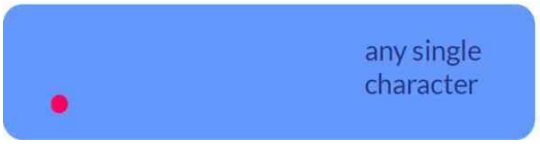
exactly

```
x = re.findall("[Tt]he ([a-z]{4})", txt)
['lazy']
```



at least X and at most Y

```
x = re.findall("[Tt]he ([a-z]{4,5})", txt)['quick', 'lazy']
```



any single character

```
x = re.findall("jumps.+dog", txt)
['jumps over the lazy dog']
```

Regular Expressions

This table shows the most frequently-used symbols.

Symbol	Description
.	Any character
\"	Double quote character
\'	Single quote character
\\	Backslash
\t	Tab character
\n	New line character
[0-9]	Any digit from 0 to 9 inclusively
[^0-9]	Any character that is not a digit
[a-z]	Any lowercase letter
[^a-z]	Any character that is not a lowercase letter
[a-zA-Z0-9_]	Any digit, any letter or an underscore

As you can see, some symbols start with a backslash. These symbols are called escape sequences and represent things that either can't be described otherwise or are reserved characters.

Now let's try to compose patterns that combine both symbols and cardinalities.

One or more digit: [0-9]+

An optional number that does not start with zero: ([1-9]{1}[0-9]+)?
Parentheses let you combine multiple symbols into symbol groups and apply cardinalities to the whole groups. You may rephrase the regular expression mentioned above the following way: zero or one number that begins with exactly one digit that is not zero and multiple arbitrary digits afterwards.

A string that may contain any lowercase letter but can't be preceded with an underscore [^_][a-z]*

In order to replace the first occurrence of a substring with another value you ought to use re.sub function. Replace dolor BLA BLA BLA amet with foobar the following way:

```
re.sub("dolor .*amet", "foobar", lorem_text)
```

Quiz

PROBLEM:

1. What is the escape sequence for a new line character?
2. What is the escape sequence for a tab character?
3. What does the re.search function return if the pattern could not be found?
4. What is going to be matched by the following call:
   re.search(r'..d', 'food bar')?
   Try to answer without the interpreter.

ANSWERS:

1. \n
2. \t
3. None
4. ood

Exercises

PROBLEM: Write a function is_valid_Python_variable_name(string) that would say if the string can be used as a variable name in Python. You don't need to worry about reserved keywords.

ANSWER:

```
1. def is_valid_Python_variable_name(string):
2.     # underscore or a letter followed by an integer,
3.     # underscore or a character
4.     found = re.match("[_A-Za-z][_a-zA-Z0-9]*", string)
5.     return bool(found)
```

Exercises

PROBLEM: Write a function get_number_of_occurrences(string, substring) that would return an integer that calculates how many instances of a substring were found in a string.

ANSWER:

```
1. def get_number_of_occurrences(string, substring):
2.     findings = list(re.finditer(substring, string))
3.     return len(findings)
```

Math module

This chapter provides an overview of the math module from the Python standard library. The module contains implementations of common mathematical operations, such as square root, that aren't inclcuded in the standard function set.

Using the math module you can perform regular trigonometric operations:

```
>>> math.sin(math.radians(30))
0.49999999999999994
>>> math.cos(math.radians(60))
0.5000000000000001
>>> math.tan(math.radians(45))
0.9999999999999999
```

Note, the argument is a value of the angle in radians not degrees. This is why an additional call to math.radians was used.

Also, you see 0.4999999999 instead of 0.5 or 0.99999999999 instead of 1.
This is due to the way the language handles floating point number calculations.

You can get special numbers, like pi and e, to a certain level of precision:

```
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

You can also calculate square roots:

```
>>> math.sqrt(16)
4.0
>>> math.sqrt(25)
5.0
```

Quiz

PROBLEM: In order to answer the following questions you need to dive into the standard library

1. What function would you use to calculate an arctangent?
2. What function should be used for exponentiation?
3. How would you calculate a factorial of a number?

ANSWERS:

1. math.atan
2. math.exp
3. math.factorial

JSON

In software engineering there is a process called serialization. The main purpose of it is to transform arbitrary object's state into something that could be transfered over the wire and/or stored in such a manner that the original object could be reconstructed later on. The process of reconstruction is called deserialization.

A real life analogy of serialization is the process of investigating an existing artifact (e.g. a car) and making a detailed blueprint of it (engine, wheels, transmission, etc). Sending serialized data over the wire is similar to sending the blueprint over a normal mail. And deserialization is the process of constructing a copy of the original object by someone based on the fetched blueprints.

In the realm of web programming two standards of data serialization gained highest popularity. XML and JSON. Both are textual which implies that data represented using these standards can be easily viewed by a human-being via a simple text editor like VIM. The subject of this chapter is JSON and the means of Python to serialize data using it.

JSON is a native way to operate with data in JavaScript programming language.

As a format JSON operates with three major constructs.

1. Primitives: integers, floating point numbers, strings, booleans and null datatype
2. Objects: equivalents of Python dictionaries
3. Arrays: equivalents of Python lists

As you can see all JSON constructs perfectly match Python built-in types. Thus the main purpose of serialization of Python data structures into JSON is to transform Python built-in types into a JSON string that could be e.g. sent as a payload inside of an HTTP request/response body.

By default Python can serialize the following datatypes: int, bool, None, str, float, long, dict and list. There are ways of extending this functionality but they are out of the scope of this chapter.

Python's module responsible for serialization is called json. It contains two major functions: **dumps** and **loads**. **Dumps** expects Python data structure as the input and returns JSON string as the output. **Loads** does the inverse - it expects a JSON string as an input and returns Python data structure as an output.

Code Examples

Import JSON module

```
1. json
   Serialize a list of strings into JSON array
2. list_of_strings = ["one", "two", "three", "four"]
3. json_array = json.dumps(list_of_strings)
```

Deserialize a JSON object into a Python data structure

```
1. json_data = """
2.    {
3.      "integer": 1,
4.      "inner_object": {
5.        "nothing": null,
6.        "boolean": false,
7.        "string": "foo"
8.      },
9.      "list_of_strings": [
10.       "one",
11.       "two"
12.     ]
13.   }
14. """
15. python_dict = json.loads(json_data)
```

Serialize an integer

```
print (json.dumps(1))
1
print (type(json.dumps(1)))
<class 'str'>
```

Code Samples

Serialize a string

```
>>> print(json.dumps("FOO"))
"FOO"
>>> print(len(json.dumps("FOO")))
5
```

Try to deserialize something. that is not JSON - ValueError is raised when the input can't be processed

```
>>> json.loads("FOO")
Traceback ...
...
ValueError: No JSON object could be decoded
```

Try to serialize something. that does not have a representation in JSON - only a limited set of Python built-in types is supported by default

```
>>> class Teacher(object):
    pass
>>> t = Teacher()
>>> json.dumps(t)
Traceback ...
...
TypeError: <...Teacher instance at ...> is not JSON serializable
```

Quiz

PROBLEM:

1. Which function is responsible for JSON serialization in Python?
2. What is the antonym of “serialization”? Which Python function is responsible for it?
3. What is the output of this command: print (json.dumps(“FOO”)[0]) ?
4. What is the output of this command: print (len(json.dumps(None))) ?
5. What is the output of this command: print (type(json.dumps(‘”1.57”’))) ?

ANSWER:

1. json.dumps
2. deserialization / json.loads
3. “1
4. 4
5. <type ‘str’>2

---

1 if you answered F - you forgot that JSON representation of a string includes quotation marks as well

2 if you answered float - you missed double quotes inside JSON string

Exercises

PROBLEM: There is the following code:

Implement join_two_arrays function

```
>>> array_of_strings_one = '["one", "two"]'
>>> array_of_strings_two = '["three", "four"]'
>>> print join_two_arrays(array_of_strings_one, array_of_strings_two)
["one", "two", "three", "four"]
>>> type(join_two_arrays(array_of_strings_one, array_of_strings_two))
<class 'str'>
```

ANSWER:

```
1. def join_two_arrays(array_of_strings_one, array_of_strings_two)
2.     list_one = json.loads(array_of_strings_one)
3.     list_two = json.loads(array_of_strings_two)
4.     joined_list = list_one + list_two
5.     return json.dumps(joined_list)
```

Exercises

PROBLEM: There is the following code:

Implement jsonize_if_not_already_json function.

```
>>> print jsonize_if_not_already_json("ONE")
"ONE"
>>> print jsonize_if_not_already_json('"ONE"')
"ONE"
>>> print jsonize_if_not_already_json(False)
false
>>> print jsonize_if_not_already_json("false")
false
>>> print jsonize_if_not_already_json("1")
1
>>> print jsonize_if_not_already_json(1)
1
```

ANSWER:

```
1. def jsonize_if_not_already_json(value):
2.     try:
3.         value = json.loads(value)
4.     except ValueError:
5.         pass
6.     return json.dumps(value)
```

SQL: sqlite3

One of the main principles of software development is reusability. The principle can be rephrased as DRY - Don't Repeat Yourself. The best way to achieve it is to avoid reinventing the wheel and apply the solutions that were implemented before you.

Over the decades of industry development in information processing and data storage, a common set of constraints has emerged. In this context, the word "constraint" means a set of rules that you apply to the data to be stored.

To understand what constraints are think about an email address. The address is expected to have an "@" character. This expectation is a constraint.

A real-life system with multiple constraints is an automobile assembly line. All the parts are expected to fit certain criteria - if they don't, it's not possible to fully automate the line, and you have to involve manual labor. A general rule of thumb is that to achieve a reasonable level of automation, you must sacrifice a fraction of the input's flexibility.

To create a common system of constraints, the existing systems were combined into what is now called a relational database.

Data are stored in the relational database as tables, where each column represents a set of rules for what data types (strings, ints, booleans, etc.) can be stored in each row.

Each column has a type that represents a rule for the database engine regarding what sort of data can be stored in respective fields. The types can be integers, strings, booleans, etc.

The primary key constraint defines which field should be used as the main index, i.e. which piece of information in the row can be used as its address. A word index means exactly the same thing as in post industry. It is a unique identifier used to simplify access to information. In case of post it is the address of a particular recipient, in case of relational database - a particular row in the table.

SQL

To illustrate the meaning of referential integrity it is the best to have a look at the following real life example.

If a student were to leave a school to attend another school, their name must be removed from the register in all of their classes. Similarly, if you want to remove an item from a relational database, you must also remove all references to it in the database.

The process of removing students from respective courses whenever they change school in the context of relational database is called referential integrity management. Foreign key constraints tell you which columns contain references to other places in the database.

There are dozens of other constraints. And even though they won't be covered in this chapter the ones that were already briefly introduced are enough to get a reasonable understanding about how relational databases behave.

The last thing you need to know about before we move on is Structured Query Language (SQL). SQL is a programming language specifically used for declaring database constraints, and for manipulating data (i.e. listing, viewing, creating, updating, or deleting data).

Python sqlite3

Python has a built-in database module called sqlite3.
It is a full stack Python implementation of a relational database and SQL.

Here we will briefly go through major SQL statements and database constructs.

To begin with, open your Python interpreter and import sqlite3

```
1. import sqlite3
   Initialize database connection:
2. conn = sqlite3.connect('example.db')
   Get a cursor:
3. cursor = conn.cursor()
4. cursor.execute("PRAGMA foreign_keys = ON")
```

Cursor is just a Python object used to execute SQL statements. In sqlite3 foreign key constraints are disabled by default for performance reasons. PRAGMA statement enables them.

Now let's create a simple users table.

```
1. cursor.execute("""
2.    CREATE TABLE users(
3.       id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
4.       email TEXT NOT NULL,
5.       name TEXT NOT NULL
6.    )
7. """)
```

In the code above you may see the following:

- every user must have an email and a name that are text values
- every user has an integer identifier that is used as a primary key. AUTOINCREMENT means that whenever new user is created id shall be generated automatically based on the identifier of a user stored previously

Also lets create a table called cars in such a way that every car mentioned in the table shall have exactly one owner from the list of humans listed in the users table.

```
1. cursor.execute("""
2.    CREATE TABLE cars(
3.       id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
4.       brand TEXT NOT NULL,
5.       model TEXT NOT NULL,
6.       owner INTEGER NOT NULL,
7.       FOREIGN KEY(owner) REFERENCES users(id) ON DELETE CASCADE
8.    )
9. """)
```

In the code above there is a foreign key definition that denotes a relationship between cars and users in a form of ownership. The ON DELETE CASCADE statement tells the database that whenever a user is removed all the cars that belong to the users must be removed as well.

Lets create one user and two cars that belongs to this user:

```
1. cursor.execute('INSERT INTO users VALUES(NULL, '
2.                '"john.smith@pythonvisually.com", "John Smith")')
3. cursor.execute('INSERT INTO cars VALUES(NULL, "Fiat", "Polo", 1)')
4. cursor.execute('INSERT INTO cars VALUES(NULL, "BMW", "X6", 1)')
```

Note, when creating cars we passed value 1 for the owner. We knew that user John Smith was the first one created in the database. Thus, his ID would logically be 1

Let's attempt to create a car that references a user who does not exist, e.g. user #2.

```
cursor.execute('INSERT INTO cars VALUES(NULL, "Mazda", "6", 2)')
```

You are expected to see a traceback with the following error message:

```
sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

This way database engine notifies you that something is wrong with the a foreign key you try to store.

Now, let's manipulate the test data that we've created.
First, let's change John's email:

The above code uses the UPDATE command and the WHERE statement. In combination, they let you update all the values that fit your search criteria.

Now, let's check that the email actually got updated:

```
rows = list(cursor.execute(
   'SELECT email FROM users WHERE name="John Smith"'))
[("john.smith@pythonvisually.com",)]
```

When choosing which user to check we used SELECT command. During selection we stated that we are interested only in user's email. In case of row selection cursor's execute method returned and iterator over database rows that we transformed into a list and printed.

Lets get rid of one of the cars, e.g. BMW:

```
1. cursor.execute('DELETE FROM cars WHERE brand="BMW"')
2. rows = list(cursor.execute('SELECT * FROM cars'))
3. len(rows) == 1
```

To remove the row we used DELETE command. After that we verified that total amount of cars was one.

And last but not least, lets check how ON DELETE CASCADE statement works. Lets get rid of Mr. John.

```
cursor.execute('DELETE FROM users WHERE name="John Smith"')
```

Now check how many cars we have left in the database:

```
1. rows = list(cursor.execute('SELECT * FROM cars'))
2. print len(rows)
3. len(rows) == 0
```

This is it. Fiat was removed from the database together with its owner.

Quiz

PROBLEM:

1. What does the acronym 'SQL' stand for?
2. What sort of integrity do foreign keys help to enforce?
3. How many different fields with the same value can exist in a primary key column?
4. SQL is used for two purposes. Name them.
5. If you want to change an existing row which command should you use?
6. In what form is data stored inside relational databases?
7. Constraints are usually applied to which part of a database?
8. Which entities do individual records in the database represent?
9. What does the acronym 'DRY' stand for?

ANSWER:

1. Structured Query Language
2. Referential integrity
3. Exactly one. Each primary key value must be unique
4. Data definition and data manipulation
5. UPDATE
6. In the form of tables
7. To an individual column
8. Rows
9. Don't Repeat Yourself

Exercises

PROBLEM: Given that the database is completely clean, write Python code that would create a user "Mike Stark" with email "subscribe@pythonvisually.com".
Add "Toyota Prius" to her collection of cars.

ANSWER:

```
cursor.execute('INSERT INTO users VALUES(NULL, `
        `"subscribe@pythonvisually.com", "Mike Stark")')
cursor.execute('INSERT INTO cars VALUES(NULL, "Toyota", "Prius", 1)')
```

Exercises

PROBLEM: Write an SQL command that would let you remove any car produced by Ford.

ANSWER:

```
1. DELETE FROM cars WHERE brand = "Ford"
```

Exercises

PROBLEM: Write an SQL command to remove every car owned by user #17.

ANSWER:

```
1. DELETE FROM cars WHERE owner = 17
```

Part III: Popular 3rd party libraries

3rd party packages

Aside from the default Python packages, there are a variety of 3rd party solutions.
In order to use them in a similar fashion as the standard library they must be placed under PYTHONPATH.
PYTHONPATH is an environment variable that tells the interpreter where to look for Python source files.

The standard way to install Python packages is to use “pip”, which has a simple command line interface.This is how to install:

https://pip.pypa.io/en/latest/installing.html

Find all packages that have a “nose” string in their name:

```
pip search nose
```

Install a package named “nose”:

```
pip install nose
```

Show information about the “nose” package.

```
pip show nose
```

Get rid of the nose package:

```
pip uninstall nose
```

Web Development with Flask

Python can be a great choice when it comes to creating websites, and there are a lot of frameworks out there to help you with this task. This chapter provides an overview of one of them, Flask (http://flask.pocoo.org/), which is very easy to get started with.

Flask is a microframework, it keeps its core simple but extensible. The base framework is intended to be very lightweight, while new features can be added through extensions. Core features are:

- built in development server and debugger
- integrated unit testing support
- heavily uses HTTP methods
- uses Jinja2 templating
- supports secure cookies
- 100% WSGI 1.0 compliant
- unicode based
- extensively tested and documented

To begin with, install the framework:

```
pip install flask
```

Hello World

Create a file name hello.py with the following code:

```
1. app = Flask(__name__)
2. @app.route("/")
3. def hello():
4.     return "Hello World!"
5. if __name__ == "__main__":
6.     app.run(debug = True)
```

and run it:

```
python hello.py
```

If you open your web browser and type:

http://localhost:5000

you should be greeted with the warming sight of "Hello World!"

Congratulations, you have just created your first website with Flask.

Let's walk through hello.py line by line.

1. First, the Flask class is imported from the flask module.
2. Then, an instance of the Flask class called app is created.
3. `@app.route("/")` tells Flask the URL that will trigger the method below it.
4. `The hello() method is defined, which returns the string "Hello World!"`
5. app.run starts the HTTP server.

Create a new file called test.py:

```
1. from flask import Flask
2. app = Flask(__name__)
3. @app.route("/")
4. def root():
5.     return "Request Path is: " + request.path
6. @app.route("/test/")
7. def test():
8.     return "Request Path is " + request.path
9. if __name__ == "__main__":
10.    app.run(debug = True)
```

You can see that a request module was imported from flask. This object represents an HTTP request sent by the client and received by the server, which contains the path of the requested page, the type of the request, and other information about the client and

the transmitted data.

In this case, you have used the request object to return the path of the page.

Flask offers an easy way to answer to different HTTP methods:

```
@app.route("/data", methods=['GET'])
def show_data():
    ....
@app.route("/data", methods=['POST'])
def handle_data():
    ....
@app.route("/user/<username>")
def greet_user(username):
    return "Hello " + username
```

You can use sections of the webpage's URL as a variable in your function.

```
import json
from flask import make_response, Flask
app = Flask(__name__)
@app.route("/json")
def get_image():
    # Create response
    response = make_response(json.dumps({"foo": "bar"}))
    # Set correct mimetype
    response.mimetype = "application/json"
    return response
if __name__ == "__main__":
  app.run(debug = True)
```

Until now, all of the pages would return simple text.

Flask would automatically transform this string into an HTTP response, and would give it the mimetype "text/html" along with a status code 200.

Now let's modify this behavior.

This code returns a JSON object.

We can also change the error code:

```
...
@app.route("/error")
def error_page():
    response = make_response("Error Page")
    response.status_code = 404
    return response
...
```

Quiz

PROBLEM:

1. What is Flask and what should you use it for?
2. Why is Flask called a microframework?
3. What is the default port for the Flask integrated HTTP server?
4. Can you cite most frequently used HTTP methods?
5. Consider the following code snippet:

```
@app.route("/error")
def error_page():
    return "Error Page"
```

ANSWER:

1. Flask is a microframework for Python used to create web applications.
2. It keeps its core simple. New functionality must be added via extensions.
3. 5000
4. GET, POST, HEAD, PUT and DELETE
5. 200. Do not be fooled by the error name. You did nothing to change the status code.

Exercises

PROBLEM:

Imagine you want to create a small discussion page for your website to let your users post and read messages. Messages are sorted in descending order, so the most recent message is displayed first.

Your website is so popular that soon there are too many messages to be displayed! You decide to limit the number of messages per page to 50, and add "next" and "previous" buttons to navigate between pages.

Can you create a script to display the first and last post number on a given page?

SAMPLE ANSWER

```
1. ...
2. @app.route("/posts")
3. @app.route("/posts/page/<int:page_nb>")
4. def show_posts(page_nb = 1):
5.     first_msg = 1 + 50 * (page_nb 1)
6.     last_msg = first_msg + 49
7.     return "Messages {} to {}".format(first_msg, last_msg)
8. ...
```

NOTES:

- set a default value for the variable page_nb so that the routes "/posts" and "/posts/page/1" are equivalent.
- force page_nb to be of type int int the decorator.
  So that a route "/posts/page/something" would return a 404 error.

Recipes for task automation

Files and Directories

Working with files and directories can rapidly grow into a tedious and time consuming task. Python offers convenient methods to help you easily walk directories and perform operations on files.

The following code will scan all directories and subdirectories in a topdown manner:

```
1. import os
2. for root, dirs, files in os.walk(".", topdown = True):
3.     for name in files:
4.         print(os.path.join(root, name))
5.     for name in dirs:
6.         print(os.path.join(root, name))
```

Slate is a small module you can use to easily extract text from PDF files.

The following code reads a PDF file and extracts all text from the document, presenting each page as a string of text:

```
>>> import slate
>>> f = open("doc.pdf")
>>> doc = slate.PDF(f)
>>> doc
[..., ..., ...]
>>> doc[1]
"Text from page 2...."
```

Python makes it easy to access remote resources using the urllib.

```
>>> from urllib.request import urlopen
>>> response = urlopen("http://a/b/c")
>>> data = response.read()
```

Note that "response" is a file-like object, which points at the remote resource.

Image Processing

The Python Imaging Library (PIL) adds image processing capabilities to Python. You can use this library to create thumbnails, convert between file formats, print images, etc.

Load image from a file:

```
>>> import Image
>>> im = Image.open("Python.jpg")
>>> data = list(im.getdata())
```

Get a list of pixels from the image opened above:

```
1. from PIL import Image
2. from urllib.request import urlopen
3. import io
4. response = urlopen("http://a/b/c.jpg")
5. im_file = io.BytesIO(response.read())
6. im = Image.open(im_file)
7. im = im.rotate(90)
8. im = im.resize((800, 600))
9. im.save("downloaded_image.jpg", "JPEG")
```

The script below downloads a picture from a given url, rotates it, resizes it, and saves it locally.

Note, io.BytesIO is used to wrap the buffer into a proper file-like object.

The following code creates a 128x128 thumbnail from a JPEG image:

```
1. from PIL import Image
2. size = 128, 128
3. filename = "image.jpg"
4. im = Image.open(filename)
5. im.thumbnail(size, Image.ANTIALIAS)
6. im.save(filename + ".thumbnail", "JPEG")
```

Directory cleanup

Let's imagine that you need to clean up a directory and remove all files from before 2002. Each file within this directory is a text document (.txt) whose first line is a date (yyyy-mm-dd).

```
1.  import os
2.  import time
3.  count = 0
4.  kept = 0
5.  deleted = 0
6.  for (root, dirs, files) in os.walk('.'):for filename in files:
7.     if filename.endswith('.txt'):
8.        f = open(filename, 'r')
9.        sdate = f.readline();
10.       f.close()
11.       pdate = time.strptime(sdate, '%Y%-m%-d\n')
12.       if pdate.tm_year > 2002:
13.          kept += 1
14.       else:
15.          os.remove(filename)
16.       deleted += 1
17.    count += 1
18. print('Total files: ' + str(count))
19. print('Kept: ' + str(kept))
20. print('Deleted: ' + str(deleted))
```

The code above does the following 4 things:

1. Imports the necessary modules.
2. Uses the "walk" method to navigate the current directory.For each text file in our directory a. Open file b. Read date using f.readline() c. Convert string date into Python object d. If date is before 2002 the file is deleted
3. For each text file in the directory:
   a. Open the file.
   b. Read the date using f.readline().
   c. Convert the date from a string to a Python object.
   d. Delete the file if the date is before 2002.
4. Keep count of the number of deleted files and display this number at the end.

Quiz

PROBLEM: Why and when would you want to automate a task?

If doc1.pdf is a PDF document whose first page is blank, what does the following code print out?

```
1. import slate
2. f = open("doc1.pdf")
3. doc = slate.PDF(f)
4. f.close()
5. print(doc[0])
```

ANSWER:

1. Every time you need to do something over and over again, you may want to think of a way to automate this task to save yourself some time and make your life easier.
2. \n\n