Intro to programming - Python

Chapter 1: Python Basics

The Basics

Common data types: Integers, Floating-points, and Strings

Data Type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, –0.5, 0.0, 0.5, 1.0, 1.25
Strings	‘a’, ‘aa’, ‘aaa’, ‘Hello!’, ‘11 things’

Math Operators

Operator	Operation	Example	Evaluates to . . .	
**	Exponent	2 ** 3	8	2^3 Andet tal opløftet
%	Modulus/remainder	22 % 8	6	Når det er blevet divideret så mange gange, som noget kan divideres, hvor meget er der så i rest
//	Integer division/floored quotient	22 // 8	2	Hvor mange hele gange kan det deles
/	Division	22 / 8	2.75	
*	Multiplication	3 * 5	15	
-	Subtraction	5 - 2	3	
+	Addition	2 + 2	4	

String concatenation and replication

Can do	Cant do
Str + Str Str * int	Str + int Str * Str

Variables (=)

To assign a value to a variable you have to use the = sign

variable = value

You can assign a variable to an empty string, by using two apostrophes “ “. (with no space between them)

variable = “”

“Truthy” and “Falsey” values

Conditions will consider some values in other data types equivalent to True and False.
When used in conditions, **0**, **0.0**, and **' '** (the empty string) are considered False, while all other values are considered True.

Restrictions:

- It can be only one word with no spaces.
- It can use only letters, numbers, and the underscore (_) character.
- It can’t begin with a number.

Valid and Invalid Variable Names

Valid variable names	**Invalid variable names**
current_balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
account4	4account (can’t begin with a number)
_42	42 (can’t begin with a number)
TOTAL_SUM	TOTAL_$UM (special characters like $ are not allowed)
hello	'hello' (special characters like ' are not allowed)

Functions

print()

The print() function displays the value inside its parentheses on the screen.

- usually wants a string

input()
The input() function waits for the user to type some text on the keyboard and press ENTER.

- always returns a string

len()
You can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

- Evaluates to an interger

str(), int(), float()
If you want to concatenate an integer such as 29 with a string to pass to print(), you'll need to get the value '29', which is the string form of 29. The str() function can be passed an integer value and will evaluate to a string value version of the integer, as follows:

- str() – converting int (or others) to a string
- int() – converting things to integers (numbers)
- float() - converting to a decimal number

Importing modules
Each module is a Python program that contains a related group of functions that can be embedded in your programs. Before you can use the functions in a module, you must import the module with an *import* statement

How to import a module

- The *import* keyword
- Name of the module
- Optionally, more module names, as long as they are separated by commas

Different modules

Name	Function	Example

sys		import sys
		
		while True:
		  print('Type exit to exit.')
		  response = input()
		  if response == 'exit':
		  sys.exit()
		  print('You typed ' + response + '.')

os		
turtle		
random	The random.randint() function call evaluates to a random integer value between the two integers that you pass it. Since randint() is in the random module, you must first type **random.** in front of the function name to tell Python to look for this function inside the random module.	import random for i in range(5): print(random.randint(1, 10))
math	The math module has mathematics-related functions	

OBS! Don't overwrite module names

When you save your Python scripts, take care not to give them a name that is used by one of Python's modules, such as *random.py*, *sys.py*, *os.py*, or *math.py*. If you accidentally name one of your programs, say, *random.py*, and use an import random statement in another program, your program would import your *random.py* file instead of Python's random module.

Performance and Big O algorithm analysis

timeit and cProfile modules

These modules measure not only how fast code runs, but also create a pro!le of which parts of the code are already fast and which parts we could improve.

Chapter 2: Flow control

Boolean values

Has only two values, that always start with capital first letter

- True
- False

Can't use as a variable name

Boolean Comparison Operators

Comparison operators, also called relational operators, compare two values

They evaluate down to a single Boolean value - True or False

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

==(equal to) and != (not equal to)

Work with values of any data type
But an integer or floating-point value will always be unequal to a string value

<, >, <=, and >= operators

Work only with integer and floating-point values.

Difference between = and ==

The == operator (equal to) asks whether two values are the same as each other.
The = operator (assignment) puts the value on the right into the variable on the left.

Boolean operators

The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value.

- and
- or
- not

The *not* operator

Needs only one Boolean value - This makes it a *unary* operator.

not	evaluates to the opposite Boolean value.	**Expression** / **Evaluates to . . .**
		`not True` / `False`
		`not False` / `True`

Binary Boolean Operators (*and*, *or*)

The and and or operators always take two Boolean values (or expressions), so they're considered *binary* operators.

and	evaluates an expression to True if ***both*** Boolean values are True; otherwise, it evaluates to False.	**Expression** / **Evaluates to . . .**
		`True and True` / `True`
		`True and False` / `False`
		`False and True` / `False`
		`False and False` / `False`

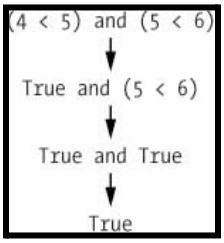
or	evaluates an expression to True if either of the two Boolean values is True. If both are False, it evaluates to False.	**Expression** / **Evaluates to . . .**
		`True or True` / `True`
		`True or False` / `True`
		`False or True` / `True`
		`False or False` / `False`

Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the and, or, and not operators are called Boolean operators because they always operate on the Boolean values True and False.

While expressions like 4 < 5 aren't Boolean values, they are expressions that evaluate down to Boolean values.



The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value.

Order of operation

Math & comparison operators → *not* operators → *and* & *or* operators

Elements of flow control

1. Often starts with *condition*
2. Always followed by *block of code*

Conditions

Almost every flow control statement uses a condition.
Always evaluate down to a Boolean value, True or False
A flow control statement decides what to do based on whether its *condition* is True or False,

Blocks of Code

Lines of Python code can be grouped together in blocks.
A block begins and ends with the indentation of the lines of code.
Three rules for blocks:

- **begin** when the indentation increases.
- can contain other blocks.
- **end** when the indentation decreases to zero or to a containing block's indentation

Flow Control Statements

All flow control statements end with a colon and are followed by a new block of code.
Always the order if, elif, (elif), else. **They are evaluated in order and as soon as one is True, it will not evaluate the rest.**
There is always exactly one *if* statement. There can be more elif statements, but they should always come after *if* and before the one *else* (if there is an *else*)
In plain English, this type of flow control structure would be

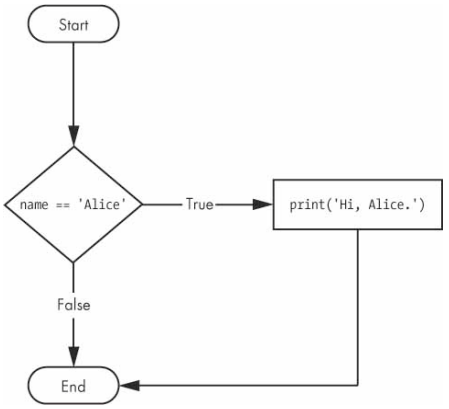
"*If* the first condition is true, do this. *Else, if* the second condition is true, do that. *Otherwise*, do something else."

if Statements

The most common type of flow control statement is the *if* statement.
All *if* statements have a **clause** (the block following the *if* statement)
If the *if* statement's condition is True, the clause will be executed. If the condition is False, the clause is skipped

An *if* statement consist of:

- The *if* keyword
- A condition (an expression that evaluates to True or False)
- A colon
- The *if* clause (an indented block of code starting on the next line)



Example:

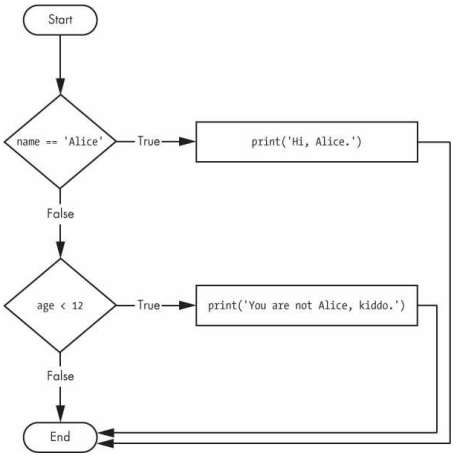
```
if name == 'Alice':
   print('Hi, Alice.')
```

elif statements

The *elif* statement is an "*el*se *if*" statement. It always follows an *if* or another *elif* statement.
It provides another condition that is checked **only** if all of the previous conditions were False.

An *elif* statement consists of:

- The elif keyword
- A condition (an expression that evaluates to True or False)
- A colon
- The *elif* clause (an indented block of code starting on the next line)



Example:

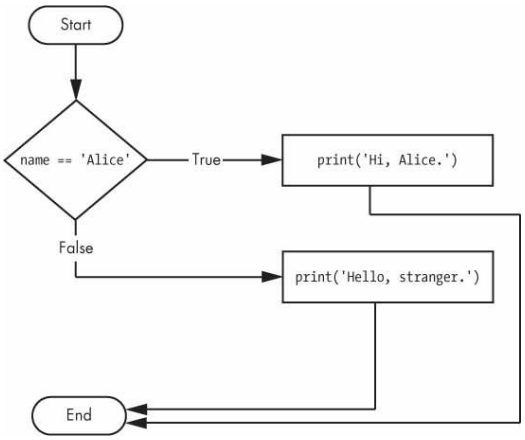
```
if name == 'Alice':
   print('Hi, Alice.')
elif age < 12:
   print('You are not Alice, kiddo.')
```

else statements

If the conditions in every *if* and *elif* statement are False, then the *else* clause is executed. It catches everything that doesn't match the *if/elif* statements' conditions and ensures that at least one (and only one) of the clauses will be executed.

An else statement never has a condition. It only consists of:

- The *else* keyword
- A colon
- The *else* clause (an indented block of code starting on the next line)

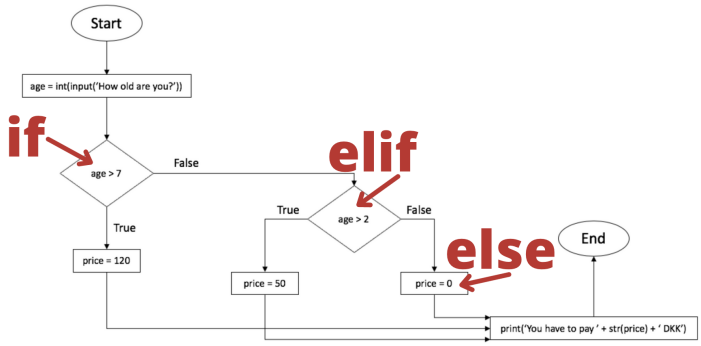


Example:

```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

OBS! Flow charts

You should ONLY use an *if/elif* statement if there is a ruby shape. Otherwise it should be an *else* (I got a mistake in mandatory assignment for making an *elif* statement instead of *else*, because the flowchart didn't have a ruby)



Loops

You can only use continue and break statements inside while and for loops.

while loop statements

You can make a block of code execute over and over again using a *while* statement.
The code in a *while* clause will be executed as long as the *while* statement's condition is True.
In the *while* loop, the condition is always checked at the start of each iteration (that is, each time the loop is executed). If the condition is True, then the clause is executed, and afterward, the

condition is checked again. The first time the condition is found to be False, the *while* clause is skipped.

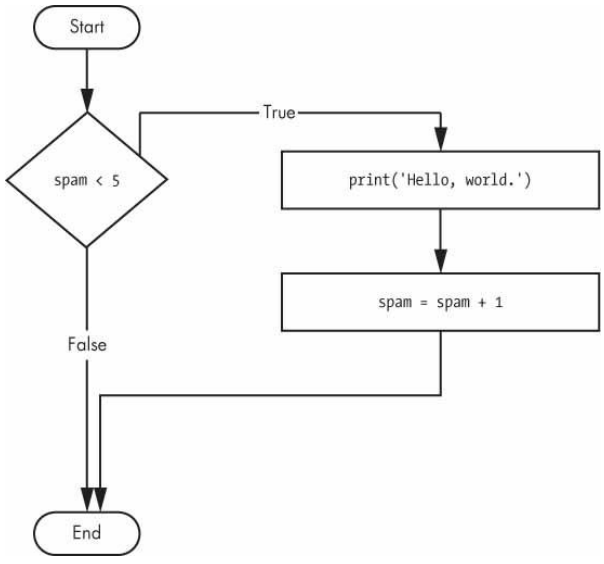
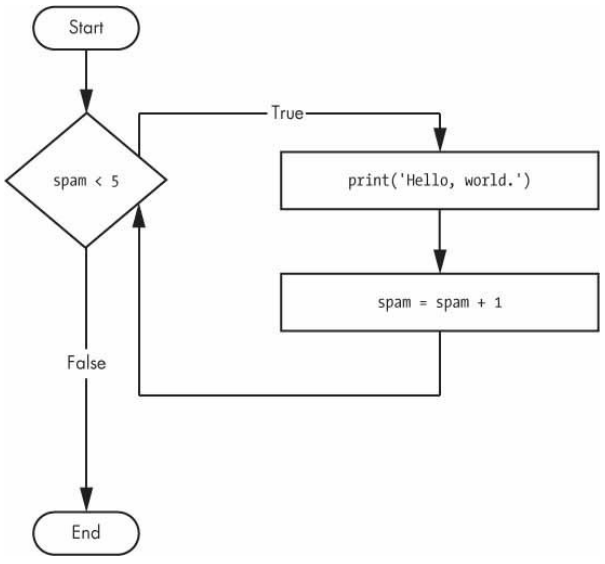
A *while* statement consists of:

- The *while* keyword
- A condition (an expression that evaluates to True or False)
- A colon
- The *while* clause (an indented block of code starting on the next line)

Difference between *if* and *while* statement

The *if* and *while* statement look very similar, but they behave differently.

if	***while***
<u>Continues after only 1 repetition:</u> At the end of an *if* clause, the program execution continues after the *if* statement.	<u>Continues as long as the condition is True</u> At the end of a *while* clause, the program execution jumps back to the start of the *while* statement.

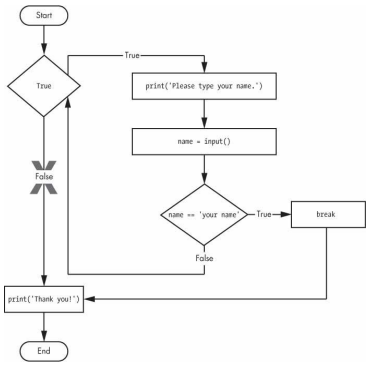
spam = 0
if spam < 5:
  print('Hello, world.')
  spam = spam + 1
 
	spam = 0
	while spam < 5:
	  print('Hello, world.')
	  spam = spam + 1
	 

Ending a *while* loop

***break* statement**

There is a shortcut to getting the program execution to break out of a while loop's clause early.
If the execution reaches a break statement, it immediately exits the while loop's clause.
A *break* statement simply contains the *break* keyword.
Will keep executing the while loop until it reaches a *break* statement

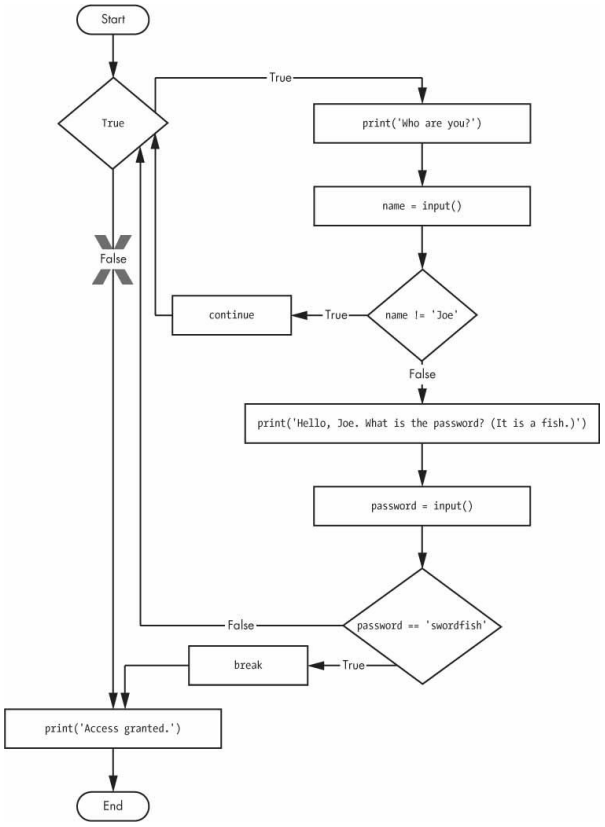
```
❶ while True:
      print('Please type your name.')
   ❷ name = input()
   ❸ if name == 'your name':
      ❹ break
❺ print('Thank you!')
```



***continue* statements**

continue statements are used inside loops.
When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.
(this usually happens when it reaches the end of the loop)
It won't get to the rest of the code, if it finds a continue

```
while True:
   print('Who are you?')
   name = input()
   if name != 'Joe':
      continue
   print('Hello, Joe. What is the password? (It
is a fish.)')
   password = input()
   if password == 'swordfish':
      break
print('Access granted.')
```



for loops and the *range()* function

If you only want to execute a block of code a certain number of times

***for in range()* statements contain:**

- The for keyword
- A variable name (eg. i or word or something else)
- The *in* keyword
- A call to the *range()* method with up to three integers passed to it
- A colon
- The *for* clause (an indented block of code starting on the next line)

Example:

```
print('My name is')
for i in range(5):
   print('Jimmy Five Times (' + str(i) + ')')
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

The first time it is run, the variable i is set to 0. The print() call in the clause will print Jimmy Five Times (0). After Python finishes an iteration through all the code inside the for loop's clause, the execution goes back to the top of the loop, and the for statement increments i by one. This is why range(5) results in five iterations through the clause, with i being set to 0, then 1, then 2, then 3, and then 4. The variable i will go up to, but will not include, the integer passed to range()

***break* and *continue* statements inside *for* loops**
You can use *break* and *continue* statements inside *for* loops
The *continue* statement will continue to the next value of the for loop's counter, as if the program execution had reached the end of the loop and returned to the start.
You can only use continue and break statements inside while and for loops.

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and *range*() is one of them. This lets you change the integer passed to *range*() to follow any sequence of integers, including starting at a number other than zero.

range(16)
- range(to)
- From 0 up to 16
- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

range(12,16)
- range(from, to)
- From index 13 to index 16
- 12, 13, 14, 15

range(0,10,2)
- range(from, to, in intervals of (eg 2))

for i in range(12,16)

Chapter 3: Functions

You're already familiar with the print(), input(), and len() functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a miniprogram within a program.

Example	Explanation
❶ def hello(): ❷ print('Howdy!') print('Howdy!!!') print('Hello there.') ❸ hello() hello() hello()	❶ The first line is a *def* statement, which defines the function named hello() ❷ The code in the block, is the body of the function This code is not executed when the function is defined, but first when the function is called, ❸The hello() lines after the function are function calls

In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses.

Execution order:

When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Purpose

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time.
In general, you always want to avoid duplicating code because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

def statements with parameters

When you call the print() or len() function, you pass them values, called *arguments*, by typing them between the parentheses. You can also define your own functions that accept arguments

❶ def hello(name): ❷ print('Hello, ' + name) ❸ hello('Alice') hello('Bob') _____	The definition of the hello() function in this program has a parameter called name. ❶ *Parameters* are variables that contain arguments. When a function is called with arguments, the arguments are stored in the parameters. The first time the hello() function is called, it is passed the

❶ def **function**(**parameter**):
  ❷ print('Hello, ' + **parameter**)
 	
❸ hello('**argument**')
  hello('Bob')
	argument 'Alice' ❸.
	
	The program execution enters the function, and the parameter name is automatically set to 'Alice', which is what gets printed by the print() statement ❷.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns.
For example, if you added print(name) after hello('Bob') in the previous program, the program would give you a NameError because there is no variable named name. This variable is destroyed after the function call hello('Bob') returns, so print(name) would refer to a name variable that does not exist.

Define, Call, Pass, Argument, Parameter

❶ def sayHello(name):
  print('Hello, ' + name)
❷ sayHello('Al')

def	The def statement defines the sayHello() function
call	The sayHello('Al') line ❷ *calls* the now-created function, sending the execution to the top of the function's code.
pass	This function call is also known as *passing* the string value 'Al' to the function
argument	A value being passed to a function in a function call is an *argument* *The argument 'Al' is assigned to a local variable named name.*
parameters	Variables that have arguments assigned to them are *parameters*. name

Return values and return statements

The value that a function call evaluates to is called the *return value* of the function

When creating a function using the def statement, you can specify what the return value should be with a return statement.

A *return* statement consists of:

- The return keyword
- The value or expression that the function should return

When an expression is used with a return statement, the return value is what this expression evaluates to.

The None value

Must be typed with capital N and represents the absence of a value

Local and Global Scope

There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call the function, the local variables will not remember the values stored in them from the last time the function was called. Local variables are also stored in frame objects on the call stack.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():
    ❶ eggs = 31337
spam()
print(eggs)
```

Chapter 4: Lists

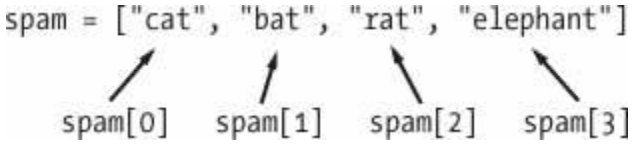
Basics

Can look like

[1, 2, 3]
['cat', 'bat', 'rat', 'elephant']
['hello', 3.1415, True, None, 42]
spam = ['cat', 'bat', 'rat', 'elephant']

Indexes

Starting with 0



Negative indexes

spam = ['cat', 'bat', 'rat', 'elephant']
spam[-1] = 'elephant'

List slicing

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

spam[2] is a list with an index (one integer).
spam[1:4] is a list with a slice (two integers)

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

List[x:y] - List[included:excluded]
X is included - takes x and all the indexes coming after.
Y is not included - takes all previous index values except y and the ones coming after.

Example

spam = ['cat', 'bat', 'rat', 'elephant']
spam[0:4]
['cat', 'bat', 'rat', 'elephant']

```
spam[1:3]
        ['bat', 'rat']
spam[0:-1]
        ['cat', 'bat', 'rat']
```

List length

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

You can use an index of a list to change the value at that index.
spam[1] = 'aardvark' means “Assign the value at index 1 in the list spam to the string 'aardvark'.”

Example

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

Lists can be concatenated and replicated just like strings.
The + operator combines two lists to create a new list value
The * operator can be used with a list and an integer value to replicate the list.

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']

>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

Working with lists

Using for loops with lists

Accessing by index instead of values - range(len(someList))

range(len(someList))

A common Python technique is to use range(len(someList)) with a for loop to iterate over the indexes of a list.

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for i in range(len(supplies)):
...     print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

Using range(len(supplies)) in the previously shown for loop is handy because the code in the loop can access the index (as the variable i) and the value at that index (as supplies[i]). Best of all, range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items it contains.

(better solution further down with enumerate)

The in and not in Operators

You can determine whether a value is or isn't in a list with the in and not in operators. Like other operators, in and not in are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value.

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
```

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

The Multiple Assignment Trick

Assigning multiple values from positions in a list

The multiple assignment trick (technically called tuple unpacking) is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition = cat
```

The number of elements in the list and the number of values have to be exactly the same.

Using the enumerate() Function with Lists

Instead of using the range(len(someList)) technique with a for loop to obtain the integer index of the items in the list, you can call the enumerate() function instead.

On each iteration of the loop, enumerate() will return two values: the index of the item in the list, and the item in the list itself.

For example, this code is equivalent to the code in the "Using for Loops with Lists":

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

The enumerate() function is useful if you need both the item and the item's index in the loop's block.

Using the random.choice() and random.shuffle() Functions with Lists

The **random module** has a couple functions that accept lists for arguments. The random.choice() function will return a randomly selected item from the list. Enter the following into the interactive shell:

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Dog'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
```

You can consider random.choice(someList) to be a shorter form of someList[random.randint(0, len(someList) – 1].

The random.shuffle() function will reorder the items in a list. This function modifies the list in place, rather than returning a new list. Enter the following into the interactive shell:

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)
>>> people
['Alice', 'David', 'Bob', 'Carol']
```

Augmented Assignment Operators

spam += 1 instead of spam = spam + 1

Augmented assignment statement	Equivalent assignment statement
spam += 1	spam = spam + 1
spam -= 1	spam = spam - 1
spam *= 1	spam = spam * 1
spam /= 1	spam = spam / 1

spam %= 1	spam = spam % 1

List- and string- -Concatenation and -replication

The += operator can also do string and list concatenation, and the *= operator can do string and list replication.

```
>>> spam = 'Hello,'
>>> spam += ' world!'
>>> spam
'Hello world!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Methods

Methods belong to a single data type. The append() and insert() methods are list methods and can be called only on list values, not on other values such as strings or integers.

Finding a Value in a List with the index() Method

List values have an index() method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn't in the list, then Python produces a ValueError error.

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
```

When there are duplicates of the value in the list, the index of its first appearance is returned.

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

Adding Values to Lists with the append() and insert() Methods

The append() method

To add new values to a list, use the append() and insert() methods.
The append() method call adds the argument to the end of the list.

```
>>> spam = ['cat', 'dog', 'bat']
```

```
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

The insert() method

The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted.

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Removing Values from Lists with the remove() Method

The remove() method is passed the value to be removed from the list it is called on.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

If the value appears multiple times in the list, only the first instance of the value will be removed.

del statement vs remove()

The del statement is good to use when you know the index of the value you want to remove from the list. The remove() method is useful when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the sort() method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

Sorted in reversed order
You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order.

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

Be aware
There are three things you should note about the sort() method.
First, the sort() method sorts the list in place; don't try to capture the return value by writing code like spam = spam.sort(). Only write spam.sort()

Second, you cannot sort lists that have both number values and string values in them, since Python doesn't know how to compare these values.

Third, sort() uses "ASCIIbetical order" rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass str.lower for the key keyword argument in the sort() method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the sort() function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Reversing the Values in a List with the reverse() Method

If you need to quickly reverse the order of the items in a list, you can call the reverse() list method.

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()
>>> spam
['moose', 'dog', 'cat']
```

Chapter 5: Dictionaries and structuring data

Basics

Like a list, a dictionary is a mutable collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

In code, a dictionary is typed with braces, {}. Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at 0 and can be any number.

```
items = {‘key1’: ‘value1’, ‘key2’: ‘value2’}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named spam would be spam[0]. But there is no “first” item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Because dictionaries are not ordered, they can’t be sliced like lists.

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary’s keys, values, or both keys and values: keys(), values(), and items(). The values returned by these methods are not true lists: they cannot be modified and do not have an append() method. But these data types (dict_keys, dict_values, and dict_items, respectively) can be used in for loops.

Accessing the values of them

Keys	Values	Items (key + value)
>>> spam = {'color': 'red', 'age': 42}	>>> spam = {'color': 'red', 'age': 42}	>>> spam = {'color': 'red', 'age': 42}

>>> for k in spam.keys(): ... print(k) color age	>>> for v in spam.values(): ... print(v) red 42	>>> for i in spam.items(): ... print(i) ('color', 'red') ('age', 42)

List from the methods

If you want a true list from one of these methods, pass its list-like return value to the list() function.

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
...  print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

You can also use the *in* and *not in* operators to see whether a certain key or value exists in a dictionary. The answer will evaluate to a boolean.

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

The get() method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

The setdefault() Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value.

Instead of this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
   spam['color'] = 'black'
```

The setdefault() method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key's value.

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time setdefault() is called, the dictionary in spam changes to {'color': 'black', 'age': 5, 'name': 'Pooka'}. The method returns the value 'black' because this is now the value set for the key 'color'. When spam.setdefault('color', 'white') is called next, the value for that key is not changed to 'white', because spam already has a key named 'color'.

To count the number of times each character occurs in a sentence

The setdefault() method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string.

```
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}
```

```
for character in message:
        ❶ count.setdefault(character, 0)
        ❷ count[character] = count[character] + 1
print(count)
```

Chapter 6: Manipulating Strings

Working with Strings

Quotes or apostrophes inside strings

String Literals

Typing string values in Python code is fairly straightforward: they begin and end with a single quote. But then how can you use a quote inside a string?

Typing 'That is Alice's cat.' won't work, because Python thinks the string ends after Alice, and the rest (s cat.') is invalid Python code.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it.

```
>>> spam = "That is Alice's cat."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character you want to add to the string.

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in Bob\'s has a backslash, it is not a single quote meant to end the string value. The escape characters \' and \" let you put single quotes and double quotes inside your strings, respectively.

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab

\n	Newline (line break)
\\	Backslash

Indexing and slicing strings

Strings use indexes and slices the same way lists do. You can think of the string 'Hello, world!' as a list and each character in the string as an item with a corresponding index.

```
 '   H   e   l   l   o   ,       w   o   r   l   d   !   '
     0   1   2   3   4   5   6   7   8   9   10  11  12
```

The space and exclamation point are included in the character count, so 'Hello, world!' is 13 characters long, from H at index 0 to ! at index 12.

```
>>> spam = 'Hello, world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[7:]
'World!'
```

Putting strings inside other strings

Putting strings inside other strings is a common operation in programming. So far, we've been using the + operator and string concatenation to do this:

```
>>> name = 'Al'
>>> age = 4000
>>> 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
'Hello, my name is Al. I am 4000 years old.'
```

F-strings

An easier way is to use f-strings

```
>>> name = 'Al'
```

```
>>> age = 4000
>>> f'My name is {name}. Next year I will be {age + 1}.'
'My name is Al. Next year I will be 4001.'
```

Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

The upper(), lower(), isupper(), and islower() Methods

The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively. Non-letter characters in the string remain unchanged.

Method	**Function**	**Example**

upper()	Returns the string as a new string but with only uppercase letters
	
	Helpful when you have to make a case-insensitive comparison
		>>> spam = 'Hello, world!'
		>>> spam = spam.upper()
		>>> spam
		'HELLO, WORLD!'

lower()	Returns the string as a new string but with only lowercase letters
	
	Helpful when you have to make a case-insensitive comparison
		>>> spam = 'Hello, world!'
		>>> spam = spam.lower()
		>>> spam
		'hello, world!'

isupper()	Returns a boolean value. Returns True if all the letters are uppercase	>>> spam = 'Hello, world!'
		>>> spam.isupper()
		False
		>>> 'HELLO'.isupper()
		True

islower()	Returns a boolean value. Returns True if all the letters are lowercase.	>>> spam = 'Hello, world!'
		>>> spam.islower()
		False
		>>> 'abcde'.islower()
		True

Obs !

They don't change the string itself. It returns a new string. If you want to update the original string you have to type

```
spam = spam.lower()
```

You can use multiple methods on some strings
Since the upper() and lower() string methods themselves return strings, you can call string methods on those returned string values as well. Expressions that do this will look like a chain of method calls. Enter the following into the interactive shell:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The isX() methods

The isX() Methods

Along with islower() and isupper(), there are several other string methods that have names beginning with the word is. These methods return a Boolean value that describes the nature of the string. Here are some common isX string methods:

isalpha()	Returns True if the string consists only of letters and isn't blank
isalnum()	Returns True if the string consists only of letters and numbers and is not blank
isdecimal()	Returns True if the string consists only of numeric characters and is not blank
isspace()	Returns True if the string consists only of spaces, tabs, and newlines and is not blank
istitle()	Returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

The startswith() and endswith() Methods

The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return False.

```
>>> 'Hello, world!'.startswith('Hello')
True
>>> 'Hello, world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello, world!'.startswith('Hello, world!')
True
>>> 'Hello, world!'.endswith('Hello, world!')
True
```

These methods are useful alternatives to the == equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The join() and split() Methods

join(): takes a list and turns it into a string

The join() method is useful when you have a list of strings that need to be joined together into a single string value. The join() method is called on a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string join() calls on is inserted between each string of the list argument. For example, when join(['cats', 'rats', 'bats']) is called on the ', ' string, the returned string is 'cats, rats, bats'.

Remember that join() is called on a string value and is passed a list value. (It's easy to accidentally call it the other way around.) The split() method does the opposite: It's called on a string value and returns a list of strings.

split(): takes a string and turns it into a list

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the string 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the split() method to specify a different string to split upon.

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

Justifying Text with the rjust(), ljust(), and center() Methods

The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
'    Hello'
>>> 'Hello'.rjust(20)
'             Hello'
>>> 'Hello, World'.rjust(20)
'        Hello, World'
>>> 'Hello'.ljust(10)
'Hello     '
```

'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

An optional second argument to rjust() and ljust() will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

Removing Whitespace with the strip(), rstrip(), and lstrip() Methods

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The strip() string method will return a new string without any whitespace characters at the beginning or end. The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively.

```
>>> spam = '    Hello, World    '
>>> spam.strip()
'Hello, World'
>>> spam.lstrip()
'Hello, World    '
>>> spam.rstrip()
'    Hello, World'
```

Chapter 9: Classes

Object oriented programming OOP

In object-oriented programming you write classes that represent real-world things and situations, and you create objects based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.

Making an object from a class is called instantiation, and you work with instances of a class.

Creating and using a Class

All classes consist of:

- The ‘class’ keyword
- The name of the class with Title case letter (fx Dog)
    - A docstring describing the purpose of the class
    - The method ‘def __init__(self, parameter1, parameter2)’ always has to contain self, but you change the parameter names so they fit the different properties (fx name, age) use as many as needed
        - A docstring under each method

- Define the parameters as such:
    - self.parameter1 = parameter1 (self.name = name)
    - self.parameter2 = parameter2 (self.age = age)

A function that's part of a class is a method. You can also give the class different other methods.

Example:

Each instance created from the Dog class will store a name and an age, and we'll give each dog the ability to sit() and roll_over():

```
❶ class Dog:
❷     """A simple attempt to model a dog."""

❸     def __init__(self, name, age):
          """Initialize name and age attributes."""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """Simulate a dog sitting in response to a command."""
          print(f"{self.name} is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(f"{self.name} rolled over!")
```

Explanation (not so important)

1 - defining the class. All classes must have Capitalized names.

2 - write a docstring describing what this class does

3 - __init__ method is a special method that Python runs automatically whenever we create a new instance based on the class. It only works with two underscores on each side. We define the __init__() method to have three parameters: self, name, and age. The self parameter is required in the method definition, and it must come first before the other parameters.

The method call will automatically pass the `self` argument.

We'll provide values for only the last two parameters, `name` and `age`.

4 - The two variables defined each have to start with 'self.', which makes them available to every method in the class, and we'll also be able to access these variables through any instance created from the class. The line self.name = name takes the value associated with the parameter name and assigns it to the variable name, which is then attached to the instance being created. Variables that are accessible through instances like this are called **attributes**.

5 - The `Dog` class has two other methods defined: `sit()` and `roll_over()`. For now, `sit()` and `roll_over()` don't do much. They simply print a message saying the dog is sitting or

rolling over. But the concept can be extended to realistic situations: if this class were part of an actual computer game, these methods would contain code to make an animated dog sit and roll over.

Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance.

1. Look at the parameters of the class __init__ method
2. Create a variable with spam = Class_name('parameter1', 'parameter2')

The variable becomes the self, so if you try to access spam.parameter1 it will give the value from self.parameter1

Example

The class Dog is a set of instructions that tells Python how to make individual instances representing specific dogs. Let's make an instance representing a specific dog:

```
class Dog:
    --snip--

❶ my_dog = Dog('Willie', 6)

❷ print(f"My dog's name is {my_dog.name}.")
❸ print(f"My dog is {my_dog.age} years old.")
```

Accessing Attributes

To access the attributes of an instance, you use dot notation. At Y we access the value of my_dog's attribute name by writing:
my_dog.name

Calling Methods

After we create an instance from the class Dog, we can use dot notation to call any method defined in Dog. Let's make our dog sit and roll over

```
my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

Setting a Default Value for an Attribute

When an instance is created, attributes can be defined without being passed in as parameters. These attributes can be defined in the __init__() method, where they are assigned a default value. Let's add an attribute called odometer_reading that always starts with a value of 0. We'll also add a method read_odometer() that helps us read each car's odometer:

```
class Car:

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
❶       self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--


❷   def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Modifying Attribute Values

You can change an attribute's value in three ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method.

Modifying an Attribute's Value Directly

The simplest way to modify the value of an attribute is to access the attribute directly through an instance. Here we set the odometer reading to 23 directly:

```
class Car:
    --snip--

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

❶ my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

Modifying an Attribute's Value Through a Method

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

```
class Car:
    --snip--

❶     def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

❷ my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

The only modification to Car is the addition of update_odometer() at X. This method takes in a mileage value and assigns it to self.odometer_reading. At Y we call update_odometer() and give it 23 as an argument (corresponding to the mileage parameter in the method definition).

Chapter 10: Files and exceptions
Chapter 13: Running time and Big O notation

O(1) and O(log n) algorithms are fast.
O(n) and O(n log n) algorithms aren't bad
$O(n^2)$, $O(2^n)$, and O(n!) algorithms are slow.

Big O measures the worst case scenario runtime a program can have

Cheat
Hvis man skal være fræk, så:
ingen for-loops = O(1) - constant
enkeltstående for-loops = O(n) - linear
for-loops inde i for-loops = O(n^2) - quadratic
looper igennem eksponentielt = O(log n)

Big O type
O(1) - constant
Runtime doesn't change with different input

O(n) - linear
Size of the worst runtime changes equally to the size of the input

for O(n) er runtime dependent på størrelsen af list
The more elements you have in your list, the longer it will take.

A linear program would look like this:
Input size(n) = 10.000 elements
Amount of time (Big O) = 10 sec

Input size(n) = 10.000 x 10 elements
Amount of time (Big O) = 10 x 10 sec

O(n^2) - Quadratic
10^2

Here the amount of time doesn't grow along with the size of the input

Bookshelf metaphor

In the following big O order examples, I'll continue using the bookshelf metaphor.

- n refers to the number of books on the bookshelf,
- big O ordering describes how the various tasks take longer as the number of books increases

Constant

Finding out **"Is the bookshelf empty?"** is a constant time operation. It doesn't matter how many books are on the shelf; one glance tells us whether or not the bookshelf is empty. The number of books can vary, but the runtime remains constant, because as soon as we see one book on the shelf, we can stop looking. The n value is irrelevant to the speed of the task, which is why there is no n in O(1). You might also see constant time written as O(c)

No matter the amount of data, the time will always be the same

Linear

Reading all the books on a bookshelf is a linear time operation. If the books are roughly the same length and you double the number of books on the shelf, it will take roughly double the amount of time to read all the books.

The runtime increases in proportion to the number of books n.

Quadratic

sorted() function takes a long time. It is O(n log n)

Notes - tilføj her:

Big O - Hjælp fra SWU'er til mandatory 5 opg 3 = O(n)

Tjek om en liste med numre er sorteret korrekt.
Hvis du har n elementer i din liste, hvad ville være det værste tidspunkt, at dit program skulle returnere false? (ville 2, 1, 3, 4 være bedre eller værre end 1, 3, 2, 4? Den ville være bedre, for den skal kun igennem for loopet en gang, og den anden skal igennem 2 gange)

Worst case runtime ville være hvis alt på nær sidste tal var sorteret, for der skulle den loope i gennem flest gange i forhold til hvor lang listen er.
så hvis man putter 8 listeelementer ind er 8 iterationer worst case

du looper også igennem lineært
for hvert element i en liste med n elementer tjekker du hvert element 1 gang
det vil sige at du tjekker n * 1 gange

O'et er big O notation
det vil sige at du tager altid det største element
så hvis den skulle løbe igennem n+2 gange
så ville det bare være O(n)
big o er bare at du fjerner alt bortset fra n'et basically
så hvis du havde en runtime på 3*n^2 +3
så ville det være O(n^2)
din liste laver faktisk 2n operations er jeg ret sikker på
men det er stadig O(n)
alt der ikke er opløftet eller logaritmet bliver bare fjernet
alt andet end n skal væk

så en funktion der lage to tal sammen
har kun en operation
hvis den tager de to tal fra en liste
har den 3 operations
en til at hente hvert objekt og en til at udføre operationen
så det ville være 3 operations
hvilket er O(1) fordi man fjerner alt - n^0 er 1

hvis man skal loope igennem et for-loop
er det generelt n operations, hvilket er O(n)

hvis man skal loope igennem et eksponentielt for-loop (f.eks. det tjekker plads 1,2,4,8,16), så er det O(Log(n))
hvis det er et for loop INDE i et for loop så er det O(n^2)
men ikke hvis det er to forskellige for-loops