Programming Lexicon

|  |  |
| --- | --- |
| Interactive shell/REPL (Read-Evaluate\_Print Loop) | platform you put prompts in |
| Program | Series of instructions |
| Mu | App in which you can interact with the interactive shell |
| Expression | Prompt/programming instruction  Consists of values and operators |
| Expression statement | **Add =** 5 + 7 |
| Values | In the expression 2+2, 2 is a Value = the thing that is going to be changed by the operator  3 different types of values:   * Integer (1, 5, 7) * Floating point numbers (1.4556) (also called floats) * Strings (anything that is not a number, or read like a text, so “3 doors” is still text) => always in “” * “” is a blank string or empty string |
| Operators | Special symbols that indicate some sort of computation should be performed  (+ is an operator => there are many of these) |
| Arithmetic operators | Basic calculator functions:  +, -, \*, /, //, %, and \*\* for math operations  + and \* for string operations |
| String Concatenation | Using an arithmetic operator on strings => combines to a single new string   * “Alice + Bob” = AliceBob   Only works on two strings |
| String Replication | Use multiply \* on string and integer value   * “Alice”\*3 = AliceAliceAlice |
| Comparison operators | We want to know if a certain comparison is true or not “Apples are equal to oranges” => FALSE  TRUE and FALSE are Boolean Values (there are only those two) |
| If, Else, Elif |  |
| Evaluate | Reduce the expression down to a single value, so 4 (when using 2+2) |
| Order of operations/precedence | Order in which the computer reads the operators (Punkt vor Strich) => use parenthesis to override this |
| SyntaxError | When Python does not understand your instructions |
| Variable | Store value in a variable (so “the variable *spam* now has the integer value *42* in it”)   * Should have a name that tells you what the value inside is = descriptive name * You cannot start a variable name with a number * No spaces or special characters * Can only be letters, numbers and underscore * Case sensitive (Spam and spam are not the same => convention to start lowercase) |
| Assignment Statements | Variable name = value to be stored  Spam=42  When you assign a new value, the old one is forgotten (overwriting) |
| Terminating a program | Stop running the program |
| Comments | Are not being read by the program, start with # |
| Commenting out code | Putting a # in front of a line of code to temporarily disable it to see what happens   * Figure out why something isn’t working |
| print() | Function that displays whatever is inside the parentheses on screen  Execution of this is called “Python is calling the print() function, and the string value is being passed to the function.” => value is now an argument   * If you put print() it will print a blank line |
| Argument | A passed value |
| input() | Function for the user to put something in it   * You can assign variables to the input * myName = input()   always returns a string, even if the user puts in a number!!! |
| len() | Evaluates the number of characters in a string   * len(“Hello”) = 5 * evaluates to an integer |
| str() | If you need an integer or float as a string value (for example to add to another string, because you can only add two integers or two strings, not integer and string)   * str(29) = “29” * print(“I am” + str(29) + “years old”) * str(2.5) = “2.5” |
| int() | Converting something into an integer   * int(1.25) = 1 * int(1.99) = 1   rounds floating numbers down   * int(“42”) = 42   useful when using input(), because input() is always a string, so:   * spam = input(101) = “101” * spam = int(spam) = 101 * this way you can do math with a string |
| float() | Convert something into a floating number   * float(“3.14”) = 3.14 * float(10) = 10.0 |
| Text and number equivalence | Integer cannot be a string, but it can be a floating point number   * 42 == “42” => false * 42 == 42.0 => true * 42.0 == 0042.0000 => true   Distinction because integers and floats are both numbers, while strings are not |
| Flow control statements | Dictates how a code is read, so which instructions are executed under which condition, like a flowchart |
| Boolean Values | True and False (always capital T and F, no “”)  Can be stored in variables |
| Comparison Operators/relational operators | Compare two values => in the end evaluate down to true or false   * == equal to * != not equal to * < less than * > greater than * <= less than or equal to * >= greater than or equal to   Example:   * 42==42 => True * 42==99 => False |
| Boolean operators | And, or, not  Used to compare Boolean values  Python evaluates the not-operator first then the and-operators, then the or-operators |
| Binary Boolean Operators | And, or  Take two Boolean values/expressions = binary   * And evaluates an expression to True if both Boolean values are True, otherwise its False * Or evaluates an expression to true if either of the Boolean values is True, only if both are False it evaluates False |
| The not-operator | Not only operates on one Boolean value = unary operator  Simply evaluates to the opposite Boolean value   * Not True = False * Not False = True |
| Flow Control | Consist of a condition and a block of code called the clause |
| Condition | Boolean expression   * (4<5) and (6<9) => expressions in this are conditions, evaluate down to True or False |
| Blocks of Code | Grouping together lines of Code  Indentation of the lines   * Blocks begin with the first indentation * Blocks can contain other blocks * Blocks end when going back to 0 or to a containing block’s indentation |
| Program execution | In Python it is just going straight down |
| Flow Control statements | Decide when what is being evaluated: if, else, elif  Order:   * If * Elif * Elif (how many you want, here order is also important * Else |
| If-Statement | Will execute if the statement is True, clause is skipped if the condition is False   * If name == “Alice”:   + Print(“Hi, Alice.”)   If clause, then everything that belongs to it must be indendet |
| Else-Statement | Can follow an if-statement => is only being evaluated if the if-statement is False  Also needs an indentation for everything that belongs to this statement |
| Elif-Statement | Mixture of else and if, always follows and if or another elif statement  Elif is the choice if all the above conditions are False  The rest of the elif-statements are skipped when the first true clause has been found   * Order matters!!!   There can be an else-statement after the elif-statements, then it is guaranteed that at least one clause will be fulfilled |
| While Loop Statements | Make a block of code execute over and over again  Will be executed as long as the while-statement condition is True   * While * Condition (expression that evaluates to True or False) * A colon * Starting on the next line, an indented block of code (called the while-clause)   Once executed, it starts from the beginning again (different to an if-clause where it just happens once, no loop)  The while loop ends only when it is False  If stuck press CTRL-C or restart shell |
| Break statements | To break out of a while loop enter the word break  For example:   * While True:   + Print(“Please type your name.”)   + Name = input()   + If name == “your name”:     - Break * Print(“Thank you”)   The loop while True is always going to be true, so you need the break to stop it  Can also be used in for loops |
| Continue Statements | Used inside a loop  When loop reaches the continue-statement it jumps back to the start  While and for loops |
| Truthy and falsey | In conditions 0, 0.0, and “ ” are considered False  All other values are considered True |
| For loop | Execute a block a certain number of times  Looks like:   * The for keyword * A variable name * The in keyword * A call to the range() method with up to three integers passed to it * A colon * Starting the next line, an indented block of code (called the for clause)   for i in range(5) |
| range() function | Used in for loops to tell them how often it should run  Can also be a range   * Range(12, 16) => numbers 12, 13, 14, 15 (up to but not including 16   Three values in a range() function: first two values are start and stopping point, third is the step argument (how much the variable increases after each iteration)   * Range(0, 10, 2) => 0, 2, 4, 6, 8   Can also be negative numbers if you want it to count minus   * Range(5, -1, -1) => 5, 4, 3, 2, 1, 0 |
| Importing modules | To import different modules:   * The import keyword * The name of the module * Optionally more module names, as long as they are separated by commas   Example:   * import random * for i in range(5)   + print(random.randint(1, 10)) |
| Random.randint() | Evaluates to a random integer value between the two integers that you pass it  Randint is in the random module, so you first have to import it and then tell python to look for the randint function in the random module, that’s why you must write random.randint |
| Sys.exit() | Terminate program early  Import sys before using it |
| Making your own function | def hello()  print(“Howdy!”)  print(“Howdy!!!”)  hello() => Howdy! Howdy!!!  def defines the new function hello()  good for grouping stuff together, that gets repeated |
| Def statements with parameters | def hello(name):  print(“Hello, “ + name)  hello(“Alice”) => Hello, Alice  defining a function = to create a function  in this case name is a parameter  and Alice is an argument |
| Parameters | Variables that contain arguments  The value stored in the parameter is forgotten once the function returns (unlike a variable) |
| Calling a function | Means to run the function (execute from the top) |
| Return value | Value that a function call evaluates to  = Result of the function  Len(Hello) => 5  5 is the return value  When creating a function you can specify what the return value should be   * the return keyword * value or expression that the function should return   looks like this:  def getAnswer(answerNumber):  if answerNumber == 1:  return “It is certain” |
| The None Value | Represents the absence of a value  None must be typed with a capital N |
| end=” “ | Deletes the spacing between words  Print(“Hello”, end=”“)  Print(“World”)   * HelloWorld |
| sep=” “ | You can fill in how you want your words to be separated  Print(“cat”, “Mouse”, “dog”, sep=”,”)   * cat,Mouse,dog |
| Local scope = local variable | Parameters and variables that are assigned in a called function exist in that function’s local scope  Can only be used in this specific local scope |
| Global scope = global variable | Variables that are assigned outside all functions  Variables must be local OR global, cannot be both  Can be accessed by different local scopes |
| Identifying errors | Try and except  If a function returns an error, put it in a try: clause and the error code in an except error: clause  def spam(divideBy):     try:         return 42 / divideBy     except ZeroDivisionError:         print('Error: Invalid argument.')  print(spam(2)) print(spam(12)) **print(spam(0))** print(spam(1)) |
| List | Contains multiple values in an ordered sequence  Looks like this: [“cat”, “dog”, “rat”]  When counting the items in the list we start at 0  Len() of the list now returns how many items are in the list |
| Items | Values inside a list  Can be “called upon” by their index number (only integer)  List[0] = “cat” |
| Lists inside lists | List = [[“cat”, “dog”], [1, 2, 3, 4, 5]  List[0]   * [“cat”, “dog”]   List[1][4]   * 5 |
| Negative indexes | Index -1 = the last index in the list |
| Slices | Von … bis in a list  List = [1, 2, 3, 4, 5]  List[0:3]   * 1, 2, 3   These slices start at the first integer and go up to but NOT INCLUDE the second integer |
| Reassigning values to indexes | You can replace the current value in a list by assigning a new one:  List = [1, 2, 3, 4, 5]  List[2] = “three”  List [1, 2, “three”, 4, 5] |
| List concatenation | [1, 2, 3] + ['A', 'B', 'C'] => [1, 2, 3, 'A', 'B', 'C'] |
| List replication | ['X', 'Y', 'Z'] \* 3 => ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z'] |
| del | Removing items from a list with del  list = [1, 2, 3, 4, 5]  del list[2]  list = [1, 2, 4, 5]  is good when you know the index of the value you want to remove (.remove() is better when you only know the value) |
| For i in range(len(someList)) | To iterate over all indexes in a list  Example:  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 |
| Determine if something is part of a list | in and not in operators  example:  list = [1, 2, 3, 4, 5]  1 in list   * True   8 in list   * False   2 not in list   * False |
| Multiple assignment trick | List = [“now”, “or”, “never”]  It, is, now = list   * Lets you assign multiple variables with the values in a list * Must be the exact same number of items, otherwise error |
| enumerate() function | Gives you the item and the index number of the item  For index, item in enumerate(list) |
| random.choice(somelist) | This way you can make a random choice out of a list including strings |
| random.shuffle() | Reorders items in a list |
| Augmented assignment operator += | += is a shortcut, with which you can add something to a variable without having to type it twice:   * Spam = 42 * Spam += 1 * Spam => 43   Instead of:   * Spam = 42 * Spam = spam +1 |
| All augmented assignment operators: | +=, -=, \*=, /=, %= |
| += | Can add to a variable  Can do list concatenation   * spam = 'Hello,' * spam += ' world!' * spam => 'Hello world!' |
| \*= | Can multiply a variable  Can string and list replication   * bacon = ['Zophie'] * bacon \*= 3 * bacon => ['Zophie', 'Zophie', 'Zophie'] |
| Method | Like a function, but is added like this: spam.**index**(“hello”) |
| index() method | Spam.index(“hello”) => if hello is in the list, the index of the value is returned so you know where it is   * spam.index(“chicken”)   If there are duplicates of a string in a list, the index of the first one will be returned  Never spam = spam.append(), always just spam.append() |
| append() method (only for lists) | Adds an argument to the end of the list |
| insert() method (only for lists) | With insert you can not only add it to the end of the list, but at any place you want to:   * spam.insert(1, “chicken”) * adds chicken at the index 1   Never spam = spam.insert(), always just spam.insert() |
| remove() method | Remove a value from a list   * spam.remove(“chicken”) => removes chicken from the list   you can only delete values that actually exist in the list  if the value appears multiple times in the list, only the first instance will be removed  good when you know the value you want to remove |
| sort() method | Sorts lists  When numbers in list it sorts it with increase in numbers  When strings it sorts it alphabetically (kinda, it also sorts upper case letters before lowercase, so Z comes before a)   * if you need actual alphabetical order: * spam.sort(key=str.lower) => this treats everything as lowercase   When you want to reverse alphabetical order:   * spam.sort(reverse=True)   can only sort only numbers or only strings, not mixed |
| reverse() method | Reverse the order of your items in the list   * spam = [1, 2, 3] * spam.reverse() * spam * [3, 2, 1] |
| \ | Split up an instruction, so it continues on the next line |
| Sequence data types | Lists, strings (a word is basically also a list of characters) range objects returned by range() and tuples  Many things you can do to lists can also be done to these other types |
| Mutable data type | e.g. lists, it can have values added, removed, changed |
| Immutable data type | e.g. strings, the single characters in a string cannot be added, removed, …  mutate a string with slicing and concatenation |
| Mutate strings | * name = 'Zophie a cat' * newName = name[0:7] + 'the' + name[8:12] * name * 'Zophie a cat' * newName * 'Zophie the cat'   The [0:7] refers to the thing we do not want changed (always 1 more than actually needed) |
| Tuple Data Type | Similar to list data type but:   * Typed in parentheses () * And they are immutable (contrary to list)   If there is only 1 value in the tuple it has to look like this:   * type(('hello',)) * double (()) and ,   communicates to other that this sequence cannot be changed |
| list() | Make a list with this function from a tuple:   * tuple(['cat', 'dog', 5]) * ('cat', 'dog', 5) * list(('cat', 'dog', 5)) * ['cat', 'dog', 5]#   Oder   * list('hello') * ['h', 'e', 'l', 'l', 'o'] |
| tuple() | Make a tuple from a list:   * tuple(['cat', 'dog', 5]) * ('cat', 'dog', 5) |
| Identity id() | Every value has its unique identity, obtain it with the id() function => returns specific number (is different each time you run this code, because computer takes the number that is currently available |
| Copy module | Has to be imported first: import copy |
| copy() | copy.copy() makes a duplicate copy of a mutable value like a list or dictionary |
| deepcopy() | copy.deepcopy() if the list contains a list |
|  |  |
| Dictionary data type | Mutable collection of many values  But here indexes can be many different data types  Indexes here are called keys => key-value pairs  Use these braces: {}  myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}   * Keys: size, color, disposition * Values for the keys: fat, gray, loud * Keys can be integers   Unordered key-value pairs  Cannot be sliced like lists  But in coding when you want to add an input (value) to a dict you do it with square brackets |
| keys() method  values() method  items() method | Can be used in a for loop to give you the content of key, value or whole items:   * spam = {'color': 'red', 'age': 42} * for v in spam.values():   + print(v) * red * 42   After using one of these methods you can form a list out of them with the list() function   * list(spam.keys()) * ['color', 'age'] |
| Checking if a key or value exists in the Dict | In and not in operators:   * spam = {'name': 'Zophie', 'age': 7} * 'name' **in** spam.keys() * True * Or: “color” in spam * False |
| The get() method | * picnicItems = {'apples': 5, 'cups': 2} * 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.' * 'I am bringing 2 cups.'   If you want to get() something you don’t have in the dictionary, it returns with 0 |
| The setdefault() method | If you want to add new item to dict:   * spam = {'name': 'Pooka', 'age': 5} * spam.setdefault('color', 'black') * 'black' * spam * {'color': 'black', 'age': 5, 'name': 'Pooka'}   Only works if there is no prior key already named color |
| pprint | First import pprint  Then with pprint() and pformat() you can make it prettier (pretty print) (a dict)  Looks like this:   * Count {} * Pprint.pprint(Count)   Especially helpful if the dict itself contains nested lists or dictionaries  If you want the pretty text as a string value instead: pprint.pformat() |
| Nested dictionaries and lists | Dict inside a dict:   * allGuests = {'Alice': {'apples': 5, 'pretzels': 12},              'Bob': {'ham sandwiches': 3, 'apples': 2},              'Carol': {'cups': 3, 'apple pies': 1}} |
| Double Quote Strings | Use double quotes “” if your string contains ‘ like: “Alice’s cat” => this way the ‘ is part of the string and does not signal that the string is over |
| Escape Character | Lets you write a character that would otherwise interfere with the string, consists of \ and the character   * “Say hi to Bob\’s mom.”   Escape characters:  \' => Single quote  \" => Double quote  \t =>Tab  \n => Newline (line break)  \\ => Backslash |
| Raw string | Raw string ignores all escape characters and prints any backslash that appears in a string  You start the string with r   * print(r'That is Carol\'s cat.') * That is Carol\'s cat.   Useful if your message contains many backlashes |
| Multiline String with Triple Quotes | Begins and ends with three single or double quotes  Any quotes, tabs or newlines are considered part of the string  Looks like this:   * print('''Dear Alice,  Eve's cat has been arrested for catnapping, cat burglary, and extortion.  Sincerely, Bob''') |
| Indexing and slicing strings | “Hello, world!” is a string, but each character has their own count, so ! is the 13th character (index 12, because we start at 0)  So, if you have a string you can do this:   * spam = “Hello, world!” * spam[0] * “H” * spam[0:5] * “Hello” |
| In and not in operators with strings | In and not in can be used with strings like with lists   * 'Hello' in 'Hello, World' * True   This is case sensitive, so it needs to be written the exact same way |
| String interpolation (putting strings inside other strings) | the %s operator inside the string acts as a marker to be replaced by values following the string  str() does not have to be called to convert values to strings  looks like this:   * name = 'Al' * age = 4000 * 'My name is %s. I am %s years old.' % (name, age) * 'My name is Al. I am 4000 years old.' |
| f-strings in Python 3.6 | Similar to %s, f prefix   * 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.' |
| upper(), lower() methods | Return a new string, where all the letters have been converted to upper or lowercase  Spam.upper(), spam.lower()   * spam = 'Hello, world!' * spam = spam.upper() * spam * 'HELLO, WORLD!   This dies not change the string itself, but returns a new string, this is why you need to use spam = spam.upper() to change the string in spam  Can also be used to take input from user regardless of their upper, lowercase use:   * print('How are you?') * feeling = input() * **if feeling.lower() == 'great':**     print('I feel great too.') * else:     print('I hope the rest of your day is good.') * this way if the user types in Great it will also count |
| isupper(), islower() | Will return a Boolean True or False, you can see if the string has all uppercase or all lowercase letters   * spam = 'Hello, world!' * spam.islower() * False |
| The isX methods | Along isupper() there are more like these methods, that all return Boolean true false:   * 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   important when you need a certain user input:   * while True:     print('Enter your age:')     age = input()     if age.isdecimal():         break     print('Please enter a number for your age.') |
| The startswith() and endswith() methods | They return true if the string value. They are called on begins/ends with the string passed to the method, otherwise they return False  Good to check if only last or first part of a string is equal to other string, alternative to ==   * 'Hello, world!'.startswith('Hello') * True |
| The join() and split() methods | Call the join() method on a list of strings, if you need to join it together with something:   * ', '.join(['cats', 'rats', 'bats']) * 'cats, rats, bats' * ' '.join(['My', 'name', 'is', 'Simon']) * 'My name is Simon'   Join makes lists to strings, split() the other way around   * 'My name is Simon'.split() * ['My', 'name', 'is', 'Simon']   The string is split where the whitespace is by default, if you want it to split somewhere else:   * 'MyABCnameABCisABCSimon'.split('ABC') * ['My', 'name', 'is', 'Simon'] * 'My name is Simon'.split('m') * ['My na', 'e is Si', 'on']   Also split something with new lines: split(“\n”) |
| The partition() method | Splitting strings along a separator and you split it in before and after:   * 'Hello, world!'.partition('w') * ('Hello, ', 'w', 'orld!')   If you pass it for o, it will only split at the first o  Can be combined with the multiple assignment trick   * before, sep, after = 'Hello, world!'.partition(' ') * before * 'Hello,' * after * 'world!' |
| The rjust() and ljust() string methods | Text appears eingerückt (with whitespace to the left or right)   * 'Hello'.rjust(10) * '     Hello'   You can also add special characters instead of whitespace   * 'Hello'.rjust(20, '\*') * '\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Hello' |
| The center() method | Centers text:   * 'Hello'.center(20) * '       Hello        ' * 'Hello'.center(20, '=') * '=======Hello========' |
| the strip(), rstrip(), and lstrip() Methods | Remove whitespace like space, tab, and newline from left, right or both sides of the 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  You can also pass it specific characters you want cut:   * spam = 'SpamSpamBaconSpamEggsSpamSpam' * **spam.strip('ampS')** * 'BaconSpamEggs'   The order of the characters passed does not matter, could be aSpm, or Spam |
| The pyperclip module | First import pyperclip  Can pyperclip.copy() and pyperclip.paste() something from you’re your clipboard |
| Regular expressions (regexes) | Like CTRL-F for searching for something, but better.  Allow for a specified pattern to be searched for  Regexes are descriptions for a pattern:  If you search for a phone number, the format would be 123-456-7899 in America, the \d regex stands for a single numeral, so if I want to search for a pattern like the phone number it would be \d\d\d-\d\d\d-\d\d\d\d or even simpler if you put numbers in {3} brackets it is like \*3 so: \d{3}-\d{3}-\d{4}  Most of them require to import the re module: import re |
| Regex objects | Passing a string value through re.compile() returns a Regex pattern object  Looks like this:   * **phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')**   it must be a raw string so r”” |
| Search for matching regex objects | You can search a string for the regex object like this:   * phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') * mo = phoneNumRegex.search('My number is 415-555-4242.') * print('Phone number found: ' + mo.group()) * Phone number found: 415-555-4242   Mo is just a generic variable name for Match objects  The group() method will return the actual matched text from the searched string |
| Step-by-step regex | 1. Import the regex module with import re. 2. Create a Regex object with the re.compile() function. (Remember to use a raw string.) 3. Pass the string you want to search into the Regex object’s search() method. This returns a Match object. 4. Call the Match object’s group() method to return a string of the actual matched text. |
| Regex tester | You can test your regex here: <https://pythex.org/> |
| Grouping regex | Create separations in regex with parenthesis () = groups  Example with the phone number you can separate the first three numbers (because they are are codes):   * (\d\d\d)-(\d\d\d-\d\d\d\d) * The first three numbers are now group 1 and the other are group 2 * Using the group() method now you can pass the group 1, 2 or nothing through * Mo.group(1) => first 3 numbers * Mo.groups() method to retrieve all groups at once (plural groups) => then use the multiple assignment trick to assign each value to a separate variable   areaCode, mainNumber = mo.groups()  if you have to search something that includes () use \ (\(\d\d\d\))  these characters have to be escaped with \: .  ^  $  \*  +  ?  {  }  [  ]  \  |  (  ) |
| The pipe | | character is called a pipe  If you have two expressions and want to match one or the other:   * heroRegex = re.compile (r'Batman|Tina Fey') * mo1 = heroRegex.search('Batman and Tina Fey') * mo1.group() * 'Batman'   Or if you want to match Batman, Batmobile, Batcopter you can write Bat just once:   * batRegex = re.compile(r'Bat(man|mobile|copter|bat)') |
| Optional matching with ? | Matching something that can potentially be there   * mo2 = batRegex.search('The Adventures of Batwoman') * mo2.group() * 'Batwoman'   The (wo)? Part is an optional group |
| Matching zero or more with the \* | \* called star or asterisk means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again   * batRegex = re.compile(r'Bat**(wo)\***man') * mo3 = batRegex.search('The Adventures of Batwowowowoman') * mo3.group() * 'Batwowowowoman' |
| Matching one or more with the Plus | + means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear *at least once*. => not optional   * batRegex = re.compile(r'Bat(wo)+man') * mo1 = batRegex.search('The Adventures of Batwoman') * mo1.group() * 'Batwoman' |
| Matching specific repetitions with braces | If you have a group that has to match a specific number of times (ha){3} matches hahaha but not haha  You can also match a range: (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'  Also match max or min: (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero to five instances |
| Greedy and Non-Greedy Matching | Pythons regular expressions are greedy by default => in ambiguous situations they will match the longest possible string  If you want non-greedy (lazy) that makes the shortest string possible, make ? after {}:   * nongreedyHaRegex = re.compile(r'(Ha){3,5}?') |
| The findall() method for regex | While search() will return a Match object of the *first* matched text in the searched string, the findall() method will return the strings of *every* match in the searched string. But only if the regex has no groups   * phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000') * ['415-555-9999', '212-555-0000']   If there are groups in the regex, it will return a list of tuples and its items are the matched strings for each group in the regex (one string for each group)   * phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups * phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000') * [('415', '555', '9999'), ('212', '555', '0000')] |
| Character classes | \d is shorthand for (0|1|2|3|4|5|6|7|8|9), but there are many more:  \d => Any numeric digit from 0 to 9.  \D=> Any character that is not a numeric digit from 0 to 9.  \w => Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)  \W => Any character that is not a letter, numeric digit, or the underscore character.  \s => Any space, tab, or newline character. (Think of this as matching “space” characters.)  \S => Any character that is not a space, tab, or newline.  No shorthand for only letters use this: [a-zA-Z] |
| Define your own character classes | Use [] to define your own character class: [aeiouAEIOU] will match any vowel   * vowelRegex = re.compile(r'[aeiouAEIOU]') * vowelRegex.findall('RoboCop eats baby food. BABY FOOD.') * ['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']   For a range of numbers or letters use hyphens: [a-zA-Z0-9]  The characters do not need to be escaped in the []: [0-5.] will match digits 0 to 5 and a period  Negative character class: placing a caret character (^) just after the character class’s opening bracket   * consonantRegex = re.compile(r'[^aeiouAEIOU]') * returns all consonants |
| Caret and Dollar Sign | caret symbol (^) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text  put a dollar sign ($) at the end of the regex to indicate the string must *end* with this regex pattern  use the ^ and $ together to indicate that the entire string must match the regex   * beginsWithHello = re.compile(r'^Hello') * beginsWithHello.search('Hello, world!') * <re.Match object; span=(0, 5), match='Hello'> |
| The wildcard character | The . is called a wildcard. It will match any character except for a newline  It will match just one character:   * atRegex = re.compile(r'.at') * atRegex.findall('The cat in the hat sat on the flat mat.') * ['cat', 'hat', 'sat', 'lat', 'mat']   Flat => lat because just one character is being matched |
| Matching everything with dot star | (.\*) stands for anything   * nameRegex = re.compile(r'First Name: (.\*) Last Name: (.\*)') * mo = nameRegex.search('First Name: Al Last Name: Sweigart') * mo.group(1) * 'Al' * mo.group(2) * 'Sweigart'   Uses greedy mode, if you want non-greedy: (.\*?) |
| Matching new lines with the dot character | The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match *all* characters, including the newline character. |
| Case-Insensitive Matching | To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile()   * robocop = re.compile(r'robocop', re.I) |
| Substituting Strings with the sub() Method | Regex cannot only find text patterns, but can also substitute new text in place of those patterns. The sub() method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The sub() method returns a string with the substitutions applied.   * namesRegex = re.compile(r'Agent \w+') * namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.') * 'CENSORED gave the secret documents to CENSORED.' |
| managing Complex Regexes | More complex regrex: telling the re.compile() function to ignore whitespace and comments inside the regular expression string, this can be enabled by passing the variable re.VERBOSE as the second argument to re.compile()   * phoneRegex = re.compile(r'''(     (\d{3}|\(\d{3}\))?            # area code     (\s|-|\.)?                    # separator     \d{3}                         # first 3 digits     (\s|-|\.)                     # separator     \d{4}                         # last 4 digits     (\s\*(ext|x|ext.)\s\*\d{2,5})?  # extension     )''', re.VERBOSE) |
| File | For something to persist after your program has finished you need to save it to a file  Has two key properties: filename and path |
| File Path | Specifies where on the computer the file is  \ are separators between folder names |
| File extension | Project.docx => docx tells you which file type it is (in this case a Word Document |
| Folders | Shown in the path, also called directories  *C:\Users\Al\Document*  The C: ist he root folder, containing all the other folders, in Windows it is C:, also called C: drive  USB drives or DVD drives can appear as new root folders like: D:\ or E:\ |
| Path() | In the pathlib module (from pathlib import Path), will return the input in a folder format |
| The / Operator to join paths | For path objects to join use /   * homeFolder = Path('C:/Users/Al') * subFolder = Path('spam') * homeFolder / subfolder * WindowsPath('C:/Users/Al/spam') * str(homeFolder / subFolder) * 'C:\\Users\\Al\\spam'   One of the first two values must be a path object |
| The Current Working Directory | Every program has a current working directory (cwd)  Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.  Get cwd:   * from pathlib import Path * import os * Path.cwd() * WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')'   Change it:   * os.chdir('C:\\Windows\\System32') * Path.cwd() * WindowsPath('C:/Windows/System32')   Can only change to a directory that does actually exist |
| The Home Directory | Everyone has a home folder:   * Path.home() * WindowsPath('C:/Users/Al')   On windows: C:\Users |
| Absolute vs. relative Paths | An absolute path, which always begins with the root folder  A relative path, which is relative to the program’s current working directory |
| Creating new folders using the oc.makedirs() function | * import os * os.makedirs('C:\\delicious\\walnut\\waffles') * this creates all of these folders |
| mkdir() method | Make a folder/directory from a Path object   * from pathlib import Path * Path(r'C:\Users\Al\spam').mkdir()   But it can only make one directory at a time |
| is\_absolute() method | This will return true if it represents an absolute path and false if it represents a relative path |
| Path.cwd() / | Get an absolute path from a relative path |
| os.path module | * Calling os.path.abspath(path) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one. * Calling os.path.isabs(path) will return True if the argument is an absolute path and False if it is a relative path. * Calling os.path.relpath(path, start) will return a string of a relative path from the start path to path. If start is not provided, the current working directory is used as the start path. |
| Parts of a file path | * The *anchor*, which is the root folder of the filesystem (C:\) * On Windows, the *drive*, which is the single letter that often denotes a physical hard drive or other storage device (C:) * The *parent*, which is the folder that contains the file (\Users\Al\) * The *name* of the file, made up of the *stem* (or *base name*) and the *suffix* (or *extension*) (spam.txt)   C:\Users\Al\spam.txt  To extract them out of the file path:   * p=Path(“C:/Users/Al/spam.txt”) * p.anchor => “C:\\” * p.parent => WindowsPath('C:/Users/Al') * p.name => „spam.txt“ * p.stem => „spam“ * p.suffix => „.txt“ * p.drive => „C:“ |
| Parents attribute | * Path.cwd() * WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37') * Path.cwd().parents[0] * WindowsPath('C:/Users/Al/AppData/Local/Programs/Python') * Path.cwd().parents[1] * WindowsPath('C:/Users/Al/AppData/Local/Programs') * … |
| File Size and Folder contents | * Calling os.path.getsize(path) will return the size in bytes of the file in the path argument. * Calling os.listdir(path) will return a list of filename strings for each file in the path argument. (Note that this function is in the os module, not os.path.) |
| Glob patterns | Simplified form of regular expressions  glob() method for listing contents of a folder according to a glob pattern  returns a generator object, that you need to pass to list() |
| Checking path validity | * Calling p.exists() returns True if the path exists or returns False if it doesn’t exist. * Calling p.is\_file() returns True if the path exists and is a file, or returns False otherwise. * Calling p.is\_dir() returns True if the path exists and is a directory, or returns False otherwise. |
| Checking if a DVD or flash drive is attached to computer | exists() method   * dDrive = Path('D:/') * dDrive.exists() * False |
| *Plaintext files* | Contain only basic text characters and do not include font, size or color info  Examples: .txt pr .py (python file)  Can be opened wit Windows Notepad |
| Binary files | All other types of files |
| read\_text() | Pathlib module so don’t forget to: from pathlib import Path |
| write\_text() method | Creates a new text file (or overwrites an existing one)   * from pathlib import Path * p = Path('spam.txt') * p.write\_text('Hello, world!') * 13 * p.read\_text() * 'Hello, world!'   These method calls create a *spam.txt* file with the content 'Hello, world!'. The 13 that write\_text() returns indicates that 13 characters were written to the file |
| Reading and writing files in Python | 1. Call the open() function to return a File object. 2. Call the read() or write() method on the File object. 3. Close the file by calling the close() method on the File object. |
| open() function | You pass the open() function a string path, so it knows where the file is, can be absolute or relative  Returns a File object (represents a file on my computer)  It can also accept strings  Opens docs in read mode => only read data from the file (cannot modify) |
| read() method | Read the contents of a file as a string value   * helloContent = helloFile.read() * helloContent * 'Hello, world!' |
| readlines() method | Gets you a list of string values from a file, one string for each line of text |
| Writing to files | You cannot write to a file that is open in read mode, instead open in “write plaintext” mode or “append plaintext” mode, or *write mode* and *append mode* for short  Write mode will overwrite the file, starting from scratch   * baconFile = open('bacon.txt'**, 'w')** * baconFile.write('Hello, world!\n') * 13   Append mode will will append text at the end of the file   * baconFile.close() * baconFile = open('bacon.txt', 'a') * baconFile.write('Bacon is not a vegetable.') * 25   If the filename passed to open() does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the close() method before opening the file again.   * baconFile = open('bacon.txt') * content = baconFile.read() * baconFile.close() * print(content) * Hello, world! * Bacon is not a vegetable. |
| Saving Variables with the shelve Module | This way the program can restore data to variables from the hard drive  Shelve module lets you add Save and Open features to your program   * import shelve * shelfFile = shelve.open('mydata') * cats = ['Zophie', 'Pooka', 'Simon'] * shelfFile['cats'] = cats * shelfFile.close()   use them to open later |
| Saving variables with the pprint.format() function | Using pprint.pformat() will give you a string that you can write to a *.py* file. This file will be your very own module that you can import whenever you want to use the variable stored in it. |
| shutil module | Has functions that let you copy, move, rename and delete files in Python programs  Import shutil, os |
| Copying files and folders | shutil.copy(source, destination) will copy the file at the path source to the folder at the path destination (source and destination can be strings or path objects)  you can also give it a new name in the process or leave the same name |
| shutil.copytree() | Will copy entire folder and every folder and file contained in it |
| Moving and renaming files and folders | shutil.move(source, destination) => moves file or folder at the path source to the path destination and will return a string of the absolute path of the new location   * move can overwrite existing files, if they have the same name, so be careful * can also rename at same time * also the folder that makes up the destination must already exist |
| Permanently deleting files and folders | Os module and shutil:   * Calling os.unlink(path) will delete the file at path. * Calling os.rmdir(path) will delete the folder at path. This folder must be empty of any files or folders. * Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted. |
| Safe delete with the send2trash module | third-party send2trash module |
| Walking a directory tree | Rename every file in some folder and subfolder, … (directory tree)  Use the os.walk() function on the directory tree  import os  for folderName, subfolders, filenames in os.walk('C:\\delicious'):     print('The current folder is ' + folderName)      for subfolder in subfolders:         print('SUBFOLDER OF ' + folderName + ': ' + subfolder)      for filename in filenames:         print('FILE INSIDE ' + folderName + ': '+ filename)      print('') |
| Compressing files with the zipfile module | Your Python programs can create and open (or *extract*) ZIP files using functions in the zipfile module |
| Reading ZIP files | First create a ZipFile object to do so: call the zipfile.ZipFile() function passing it a string of the .ZIP files filename |
| Extracting from ZIP files | The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file |
| Creating and adding to ZIP Files | To create your own compressed ZIP files, you must open the ZipFile object in *write mode* by passing 'w' as the second argument |
| openpyxl | Allows you to read and modify Excel  3rd party have to install beforehand  import openpyxl |
| Workbook | Excel spreadsheet document  A single workbook is saved with the .xlsx extension  Can contain multiple worksheets  The current or last viewed sheet is called active sheet  Sheet has columns (A, B, …) and rows (numbers), a box in a particular row and column is called a cell |
| openpyxl.load\_workbook() | >>> import openpyxl >>> wb = openpyxl.load\_workbook('example.xlsx') >>> type(wb) <class 'openpyxl.workbook.workbook.Workbook'> |
| sheetnames  getting sheet names | >>> import openpyxl >>> wb = openpyxl.load\_workbook('example.xlsx') >>> wb.sheetnames # The workbook's sheets' names. ['Sheet1', 'Sheet2', 'Sheet3'] |
| Getting cells from a sheet | First get a worksheet object, then access a cell object by name  >>> import openpyxl >>> wb = openpyxl.load\_workbook('example.xlsx') >>> sheet = wb['Sheet1'] # Get a sheet from the workbook. >>> sheet['A1'] # Get a cell from the sheet. <Cell 'Sheet1'.A1> >>> sheet['A1'].value # Get the value from the cell. datetime.datetime(2015, 4, 5, 13, 34, 2) |
| Cells and values | The Cell object has a value attribute that contains the value stored in that cell. Cell objects also have row, column, and coordinate attributes that provide location information for the cell. |
| openpyxl.utils.column\_index\_from\_string() function | For converting from letters to numbers  >>> import openpyxl >>> from openpyxl.utils import get\_column\_letter, column\_index\_from\_string |
| openpyxl.utils.get\_column\_letter() function | To convert from numbers to letters  >>> import openpyxl >>> from openpyxl.utils import get\_column\_letter, column\_index\_from\_string |
| Getting Rows and Columns from the Sheets | slice Worksheet objects to get all the Cell objects in a row, column, or rectangular area of a spreadsheet  Then you can loop over them  To access the values of cells in a particular row or column, you can also use a Worksheet object’s rows and columns attribute. These attributes must be converted to lists with the list() function before you can use the square brackets and an index with them |
| Functions, methods and data types involved in reading a cell out of a spreadsheet file | 1. Import the openpyxl module. 2. Call the openpyxl.load\_workbook() function. 3. Get a Workbook object. 4. Use the active or sheetnames attributes. 5. Get a Worksheet object. 6. Use indexing or the cell() sheet method with row and column keyword arguments. 7. Get a Cell object. 8. Read the Cell object’s value attribute. |
| CSV | Comma-separated values  Simplified spreadsheets stored as plaintext files  Python has a csv module  Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. => the simplicity is their advantage, it is a text file with comma separated values  import csv |
| Reader objects | Lets you iterate over lines in a csv file  ➊ >>> import csv ➋ >>> exampleFile = open('example.csv') ➌ >>> exampleReader = csv.reader(exampleFile) ➍ >>> exampleData = list(exampleReader) ➎ >>> exampleData  Then you see the data  Don’t just open, pass it through the reader, then convert to list, then store in example variable |
| exampleData[row][col] | Lets you access a particular row and column |
| Large files | Use the reader object in a for loop |
| Writer object | Lets you write data to a csv file, use the csv.writer() function     >>> import csv  >>> outputFile = open('output.csv', 'w', newline='')  >>> outputWriter = csv.writer(outputFile)    >>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])    21    >>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])    32    >>> outputWriter.writerow([1, 2, 3.141592, 4])    16    >>> outputFile.close() |
| change the delimiter and line terminator characters | The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the delimiter and lineterminator keyword arguments with csv.writer().  If you change the commas to tabs for example the file will be a .tsv (tab-separated values) |
| DictReader and DictWriter CSV Objects | For CSV that contain header rows  The reader and writer objects read and write to CSV file rows by using lists. The DictReader and DictWriter CSV objects perform the same functions but use dictionaries instead, and they use the first row of the CSV file as the keys of these dictionaries. |
| JSON | Stores information as JavaScript source code in plaintext files  JSON = JavaScript Object Notation  Used in many web applications  A way to format data as a single human-readable string, resembles what a Python pprint() function does |
| API | JSON content is a way for programs to interact with websites = providing an application programming interface (API)  With APIs you can:   * Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.) * Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook. * Create a “movie encyclopedia” for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer. |
| The json module | To interact with python and JSON  Can only contain following python data types: strings, integers, floats, Booleans, lists, dictionaries, and NoneType |
| Reading JSON with the loads() Function | To translate a string containing JSON data into Python value, pass it through json.loads() function  >>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0, "felineIQ": null}' >>> import json >>> jsonDataAsPythonValue = json.loads(stringOfJsonData) >>> jsonDataAsPythonValue {'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None} |
| Writing JSON with the dumps() Function | Translates a Python value into a string of JSON-formatted data  >>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None} >>> import json >>> stringOfJsonData = json.dumps(pythonValue) >>> stringOfJsonData '{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }' |

Short summary of Regex Symbols:

* The ? matches zero or one of the preceding group.
* The \* matches zero or more of the preceding group.
* The + matches one or more of the preceding group.
* The {n} matches exactly *n* of the preceding group.
* The {n,} matches *n* or more of the preceding group.
* The {,m} matches 0 to *m* of the preceding group.
* The {n,m} matches at least *n* and at most *m* of the preceding group.
* {n,m}? or \*? or +? performs a non-greedy match of the preceding group.
* ^spam means the string must begin with *spam*.
* spam$ means the string must end with *spam*.
* The . matches any character, except newline characters.
* \d, \w, and \s match a digit, word, or space character, respectively.
* \D, \W, and \S match anything except a digit, word, or space character, respectively.
* [abc] matches any character between the brackets (such as *a*, *b*, or *c*).
* [^abc] matches any character that isn’t between the brackets.

Reading chapters:

1-7, 9, 10, 13, 16

Some links:

<https://stackoverflow.com/questions>

add "site:stackoverflow.com" to your Google query

<https://www.w3schools.com/python/python_ref_string.asp>