Tutorials

April 29, 2020 2:29 PM

Download Python
https://www.python.org/downloads/

Code Examples
https://wiki.python.org/moin/BeginnersGuide/Examples

Step 1: The tutorial
https://docs.python.org/3/tutorial/

Tutorial recommended by Tiago
https://cs231n.github.io/python-numpy-tutorial/?fbclid=IwAR3fv4dFqhT-K916116XQ6G6w5mifajwK0ebFag_u970WE-VMkNT7PZz0GU

Scripting Environments (IDE's)

April 29, 2020 2:28 PM

This official Python page includes many.
https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

Tiago recommended using Anaconda and VS Code:
Anaconda:
https://www.anaconda.com/products/individual
https://get.anaconda.com/distribution/tutorial/
Visual Studio Code:
https://code.visualstudio.com/docs/languages/python

Name	Where it Runs	Details
Google Collaboratory	In browser	• Installation free • Cloud computing • "Like a Jupyter notebook stored on Google Drive" • Code is entered and run in cells • Text cells provide a place to comments that can have rich formatting • Can search built-in library of code snippets • http://research.google.com/seedbank -> A bank of many workbooks for a wide variety of applications which you can view, run • Can choose what to run on: GPU's or TPU's

Syntax

April 29, 2020 2:28 PM

The complete syntax guide:
https://docs.python.org/3/reference/
The complete standard library: (functions etc)
https://docs.python.org/3/library/

The language is interpreted as opposed to compiled. That means that the code gets compiled at run time. With a compiled language, the code that *you* write is compiled once, into machine code - all bits. That machine code itself is distributed to customers etc. as .exe or whatever format. Whereas an interpreted language gets interpreted every time it is run. That means that you ship out the code, with an interpreter that is specialized for the hardware or OS. Then each computer that runs it will compile it to machine code, (0's and 1's) before running it, every time.

*Important: multiple assignments can be made by using commas on either side of the assignment operator '='
Example: For Fibonacci sequence, redefine for the next steps by doing **a, b = b, a+b** -> what was b gets assigned to a, and b+a goes to b
This can also be done with different data structures such as a list. x[0], x[1] = x[1], x[0]

Key Word(s)	Usage
For [iterator term] **in** [iterable object]:	The for loop! Explained at this webpage: https://towardsdatascience.com/python-basics-iteration-and-looping-6ca63b30835c Essentially, the idea is that python breaks away from the 'initialize, end condition, change on iterate' paradigm by instead having an object outlook - the looping occurs over a collection of objects. That's why the latter argument must be an *iterable* object- it yields a collection of items to be iterated over. The iterator term is simply the variable to be used in the loop code More looping strategies from python documentation https://docs.python.org/3/tutorial/datastructures.html#tut-loopidioms
None	Defines null value, no value at all. This is *not* the same as 0, False, or empty string. It's a data type of its own (NoneType) and only None can be None. I.e. anything compared against it using 'is' will be false unless the object of comparison is none This is returned by a function when it has no *return* statement
Is	Test two object identities against one another which can be used to identify when there is naming redundancy
False/True	• Truth *values* equal to 0 and 1 respectively (when the value is used in math computation). They're **not** independent types, rather just values, of 'bool' value (for boolean). • Any non-zero integer value is true
And	A logical operator unto itself. Only returns true if both logical expressions on its left and right sides are also true
As	Creates an alias to which something is assigned for reference. A matter of convenience when importing something, as assignment operator cannot be used with import. So that when that library has to be referred to, the shorter alias can be used
Assert	A useful tool for debugging. It requires a logical expression follow it, for example x==1 If the expression is not true, an assertion error will be raised. A particular message can be printed in this case by putting a string after a comma
Async/Await	A pretty significant rabbit hole regarding threading/multi-processing etc. Leaving for future work but read this interesting tutorial to get a preliminary understanding to chew on: https://realpython.com/async-io-python/
Yield	Replaces the '*return*' keyword in a function. If *Yield* is used instead of *Return*, the function is automatically a generator function. When the *yield* word is used, the function gets suspended once its reached, returning the specified expression. When called again, execution resumes right where it left off. Thus the function itself is a generator, computing a value each time it is called *Must understand iterables and generators before understanding this keyword:* *https://pythontips.com/2013/09/29/the-python-yield-keyword-explained/* *Iterables: Anything you can use 'for …. in …' on. All values within are stored in memory* *Generators: Iterables that can only be iterated over once. When doing so, values are not stored in memory, they're generated when needed* ***Yield**: It's used in lieu of the return keyword. The yield keyword can be:* • *enacted inside of a loop, such that the generator returned will in total yield the same number of elements as the number of loops that could be gone through in creating it* • *Enacted outside of a loop, where some other program structure makes it's that the desired output is reached (and identified by yield) as the program is worked through* --- *Useful for when your loop will return large volumes of data that need only be iterated through once, as the generator will be less costly than having it all in memory.*
While else	• Simply have a logical expression follow the *while* statement follow by a colon ':' • Then the code that is to be looped is indented as appropriate and that's all • The *else* code block will be evaluated *after* the condition is no longer true. • The *else* code block will not evaluate if *break* is used to exit the loop
if: ***elif:*** ***else:***	Follow each (except else) by a logical test and colon':' 'elif' and 'else' are both optional
Break	Exits the *innermost* enclosing for or while loop.
In	Tests whether or not a sequence contains a value Eg. If '5' in [1,2,3,4,5]: etc

Functions

Topic	*Details*
Lambda functions	Python uses lambda functions-> A function that is defined on the fly in the form: Lambda [arg1,arg2,…,argn] : [A function of the arguments] This function can be assigned to a variable, or it could be entered in isolation like this into, for example another function input
As arguments	Methods are also objects, in that like a lambda function they can be passed as an argument. Tiago's SigmaSum function showed that, as does the example on this page: https://towardsdatascience.com/python-basics-iteration-and-looping-6ca63b30835c where a loop is *manually* defined using the objects and process that the *for*.. *In* keywords enact.
Creating a function	Format: **Def** {FuncName}(arg1,arg2,…):
Arguments they take	• Arguments come in two forms, 'positional' and 'keyword' 1. **Positional Args:** These arguments are not assigned default values. They should be the first arguments specified in the function definition. The user must enter the inputs in the sequence according to the position number of each argument, i.e. the sequence in which the function was defined. 2. **Keyword Args:** Generally these are arguments that are optional, they should be assigned default values which they'll take on if nothing was passed In for a given argument. That is done by assigning it within the function definition like Def myFun(arg1=0). When Keyword (i.e. optional) arguments are used, they must appear after the required arguments in the argument list. **a.** When the function is called, a value can be assigned to the keyword argument by name, or

	it can simply be entered into the correct sequence (w.r.t. position) **b.** These don't need to be assigned in order, the keyword assignments can come in whatever order 3. Hybrid: The third class of parameters are those that can be passed either by positional or by keyword. To be as strict as possible, they are differntiated with the optional '/' and '*' arguments a. Example: def func(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2): b. Following from this, the '/' or '*' can also be used just one or the other to identify that parameters must be entered positionally, or by keyword, depending on where the character is placed **4. Variadic Arguments:** https://medium.com/understand-the-python/understanding-the-asterisk-of-python-8b9daaa4a558 Used when the number of arguments that will be entered is not known. There are two forms of variadic input. The variadic input for positional arguments, and that for keyword arguments. An arbitrary number of inputs of either form is captured by variable *'packing'*. They go into the associated catch-all variables. The catch all variable can take whatever name, as long as it is preceded by * or ** for positional and keyword arguments respectively. Example: def myFun(*PosnArgs,**KeyWordArgs) **a.** The entered variables are stored in a *tuple* (for positional arguments) and a *Dictionary* (for keyword arguments) i. The variables *do* not need to be entered as such. That is, multiple separate variables or whatever can be inputted as if there were an argument designed to take them in. ii. The **name* argument will become a single tuple containing all the extra positional arguments (simply those without a keyword attached) iii. The ***name* argument will become a dictionary containing all the key : value pairs entered b. As mentioned in (a.i.) above, the extra arguments are just added to the function call as if there were arguments defined for them. However, the extra arguments can also be w=passed in as a tuple or list with the '*' suffix for each element in the list to be treated as a different positional argument (as opposed to the entire list being treated as one argument) **c.** If a bare asterisk is entered as an argument, '*', it forces all arguments to be entered as keyword arguments.
Runs in full	Functions, when, called (excepting functions that use *yield* (see above)) run in full from start to finish. The implication is that there can be other command beyond the *return* (or *yield*) keywords
Documentation String	If the line immediately after the function definition is a string literal on its own, it's a doc string. Use it to describe the function
Default Values	• If a default value is set as equal to a variable, it will be evaluated at run time of the function definition! • Example: i=1 def myFun(x=i) Return x+5 i=2 myFun • The above function will return 6 even though i=2 when it is called because it was defined at i=1 • Default values are only evaluated once. • If a functions default value is changed within its own scope, then that changed value will persist as the default in future iterations unless it is overwritten • To prevent this behaviour, set the default equal to none, which a user would never enter since it has no meaning. Then have an if statement checking if that argument is none, and if it is, to set it equal to the desired default

Data Types

April 29, 2020 2:28 PM

Copies of data

- Shallow: The data is copied by *reference* to the source
- Deep: The data is really copied from the source, and is stored at a different place, without making further reference to the source

Python is very OOP. So far as I've seen so far, even functions are objects unto themselves

<table>
<tr><th>Idea</th><th>Details</th></tr>
<tr><td>Dynamically Typed</td><td>• No variable declarations!
• Ties into the fact that its interpreted. If it were being compiled, the program would need to contain the declarations to ensure enough memory was stored for running. But because it is interpreted line by line, it doesn’t need to do that. It just sticks a value in memory when the time comes, then labels it</td></tr>
<tr><td>Mutability</td><td>Some collection classes are mutable. The methods that add, subtract, or rearrange the collections members, and don't return a specific item, never return the collection instance itself, but return 'None'</td></tr>
<tr><td>Object Testing and manipulation?</td><td>Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the repr() function or the slightly different str() function). The latter function is implicitly used when an object is written by the print() function.</td></tr>
<tr><td>Truth Testing</td><td>By default, an object is considered true unless its class defines either a __bool__() method that returns False or a __len__() method that returns zero, when called with the object. 1 Here are most of the built-in objects considered false:
• constants defined to be false: None and False.
• zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
• empty sequences and collections: '', (), [], {}, set(), range(0)
Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated. (Important exception: the Boolean operations or and and always return one of their operands.)</td></tr>
<tr><td>Booleans</td><td>These are the Boolean operations, ordered by ascending priority:
<table><tr><th>Operation</th><th>Result</th><th>Notes</th></tr><tr><td>x or y</td><td>if x is false, then y, else x</td><td>(1)</td></tr><tr><td>x and y</td><td>if x is false, then x, else y</td><td>(2)</td></tr><tr><td>not x</td><td>if x is false, then True, else False</td><td>(3)</td></tr></table>Notes:
1. This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
3. not has a lower priority than non-Boolean operators, so not a == b is interpreted as not (a == b), and a == not b is a syntax error.</td></tr>
<tr><td>Comparisons</td><td>Chaining works! 0<x<10 is valid, evalutes the two contained expressions (i.e. 0<x AND x<10)
<table><tr><th>Operation</th><th>Meaning</th></tr><tr><td><</td><td>strictly less than</td></tr><tr><td><=</td><td>less than or equal</td></tr><tr><td>></td><td>strictly greater than</td></tr><tr><td>>=</td><td>greater than or equal</td></tr><tr><td>==</td><td>equal</td></tr><tr><td>!=</td><td>not equal</td></tr><tr><td>is</td><td>object identity</td></tr><tr><td>is not</td><td>negated object identity</td></tr></table>Objects of different types (except numerics) never compare equal. '==' is always defined but for some object types, is equivalent to 'is'. The greater/less operators are only defined where they make sense i.e. Not for complex numbers</td></tr>
</table>

'Special Methods'	When a method name is surrounded by '__' (as seen in an example above) it's typically just so that these names don't conflict with user defined names, as the methods with these are very often-desired names.
Generators	Made using round brackets () instead of, for example, [] which would make a list. See the Syntax tab for more info on generators with *Yield* keyword

Types	**Details**
Numerics	Int, float, complex. Boolean is int sub type. Flots are usually implemented with Doubles in C. Casting can be done with int(), float(), complex()
Sequences	*Generally*: Contains data elements, they are ordered sequentially -Has 0 base indexing and allows slicing -Lists, tuples, and range (some other unique types as well) -Can use keywords *n, not in*
Mappings	*Generally*: maps hashable values to arbitrary objects. Mutable Only one type: the *dictionary*
Classes Instances	
Exceptions	

Data Structures

April 29, 2020 2:28 PM

Refer to the Standard Library for full list of non-essential built-in object types, functions, and modules

NOTE: the dir(*object*) method returns a list of all the methods an object has which could be very handy for exploring

Structure	Details	Methods
List ---- [1,2,3] etc [var1,var2] etc	• 'most versatile' • Used to group values • Written as comma separated values (or items), inside of square brackets • **CAN** contain objects of different types • Can be indexed and sliced like all sequence-types • Support concatenation which just puts them back to back, again like a string • Are mutable: individual contents can be changed • [] is an empty list • Assignments can be made to slices • Assigning an empty list will eliminate the list element • Lists within lists use layered indexing x[0][1] • Using the list(…) method on an iterator object will yield a list containing all the iterators elements. Same goes for a generator.	• .remove*(element)* -or removes an element where it is • .insert(*index, element)* the new element will take on the specified index, the others will be shifted over • .append*(object)* - adds an *object* to the list (list increases by 1) • .extend*(iterable)* -incorporates an iterable of size n into the list (list increases by n) • .reverse() flips the sequencing of all the elements • .pop(*index*) removes an item from the list, which the function returns • .sort(*key=…,reverse=False)* transforms the list to make it sorted. *Key* can be a function applied to each element which provides the basis on which to sort. E.g. len . Reverse is boolean to reverse order • Sorted(*list*,…) Function, not a method! List is first arg, other 2 are same as .sort. **Returns an iterable list** • .index(*element*) takes in a stored value and returns the index where its stored • .count*(element*) returns the count of how many times an element appears in a list *Note: for list size, use len(list)*
Tuple --- () empty tuple a, or (a,) a, b *same as* (a, b) tuple()	• Immutable! • Used where an immutable sequence is needed, such as for storing in a set or dictionary • Note how the comma is used to create a singleton tuple • Parentheses optional for listing items into a tuple • Using the tuple() on an iterable creates a tuple with the same items in the same order as the iterable. If it takes a tuple, it stays unchanged • It is the comma that makes the tuple not the parentheses	• .count(*element*) returns the count of how many occurrences there of an element within a tuple • .index(*element*) returns the smallest index number at which an element appears in the tuple *These are the only methods custom to tuple type objects, see list of functions that work on any iteratable)* *Special Feature: tuple to tuple multi-assignment*

Generator/Iterator	• When an iterable gets iterated on, an iterator object is created. Sometimes it's a generator object
	  • The generator object generates a number or value using some function. The iterator simply works through some discrete list.
	• A characteristic feature of these is that as they are iterated through, they cannot be reverse-iterated.
	  • Iterator objects cannot retrieve	

	old data one it's been passed over • Generator objects functions cannot go back in reverse	
Dictionary --- {k1:v1} {k1:v1 , "k2":v2} etc Dict(***kwarg*) Dict(*mapping, **kwarg*) Dict(*iterable, **kwarg)* Iter(*dict'nry)* returns an iterator over the keys	• The dictionary is not a set or a collection, because those words refer to other structures. It is a *mapping* • It maps {key : value} pairs to one another • Keys are almost arbitrary. The only constraint is that they be hashable • An object is hashable if it has a hash value which never changes during its lifetime: it has a __hash__() function, and can be compared to other objects, i.e. it has an __eq__() method • Therefore, they can't be lists, dictionaries, or other mutable types that are compared by value rather than identity • Constructed of key : value pairs separated by commas, all within braces • When the dict() method is used with an iterable within it, each item in the iterable must itself be iterable with exactly 2 objects. The first will always become the key, and the second, the value. If a key occurs more than once, the last value that appears in the process is the one that's taken on (i.e. overwriting occurs) • To all a value, can use an index style approach [*key*]	
Set --- { v1, v2,…} etc	• An unordered collection of distinct hashable objects • Compatible with ***in***, *len(), for* • *Not compatible with sequenced operations such as slicing* • Generally used for membership testing, removing duplicates from a sequence, and computing math operations such as intersection, union, difference, symmetric difference	
Range --- Range(*stop)* Range(*start, stop, step)*	• Treated as a sequence when comparisons performed • Always takes the same amount of memory no matter how large it is (a generator I suppose) • Effective way to build large set of numbers	

Functions that can be applied to any iterable

Name	Description
Casting functions	The functions list(), tuple(), can be applied to *any iterator* to take the iterators outputs into a list or tuple. Dict() can only be used when each output from the iterator is an iterable with 2 elements

	that conform to the other constraints of the dict type such as key hashability. The enumerate() iterable output is an example of one that works for the dictionary type.
Any*(iterable)*	Returns true if any element is true
All*(iterable)*	Returns true if all elements are true
Enumerate*(iterable)*	Returns a generator object that creates a **tuple** of (incrementingid, element) for each element in the iterable
filter*(function, iterable)*	Constructs an **iterator** from elements of an iterable for which the assigned function returns true. *If no function is entered, it defaults to the Identity/ all() function, returning false if any elements are false)*
max*(iterable)*	Returns the largest item
min*(iterable)*	Returns the smallest item
Map*(map_function,iterable)*	Returns an iterator which iterates over the iterable, applying the map function to each element
Reversed*(iterable)*	Returns *an iterator* which iterates over the elements of the iterable in reverse
Sorted(*iterable*,*key=…, reversed=False)*	Returns a *list* which contains the sequence of elements from the iterable that is sorted. Optional parameters reversed and key can be used to modify outcome. Key is a function that gets applied to every element, then all elements would be sorted by their outputs.
Sum(*iterable)*	Returns the sum of items In an iterable
zip*(iterable)*	Returns an iterator object that creates tuples which contain the elements from each iterable collected by like index. i.e. a tuple containing all the first elements, the following tuple containing all the 2nd elements, etc.

Libraries

April 29, 2020 2:29 PM

A website for storing code snippets - There will be a code example for basically anything you want to do
http://code.activestate.com/recipes/langs/python/

Keep Code Compatible
In Anaconda, you can define 'environments' -so code you write will always be run with the packages that it was coded on. That environment will not take on package updates, ensuring code remains functional and doesn't break. Within the environments tab simply select which packages and versions to use.

DEFINITIONS
Module- Python code contained in a file (any file….)
Package- Python codes contained in a directory of file(s)
Library- A collection of modules and packages

Colloquially/in practical usage, can think of libraries/packages as collections of custom code that are distributed for a particular purpose (for example, a data analysis package containing all the custom functions etc. that do data analysis)

Library	Application	Features	Documents
Pandas	Data Analysis	• Data Frame object for data manipulation with integrated indexing • Tools for reading and writing data from memory to different formats like CSV, text,Excel, SQL, etc. • Means for handling messy data • Flexible pivoting of data sets • Aggregating or transforming data with split-apply-combine operations	https://pandas.pydata.org/docs/

https://gym.openai.com
"A toolkit for developing and comparing reinforcement learning algorithms"
-Tiago used/recommends it

Runtime Behaviour

May 4, 2020 3:08 PM

Behaviour	Details
Variable Referencing	When a variable is called on, first the code draws on the local function table, and progresses out through the tables of nested functions, then the global symbol table, and finally the built-in name table.