runtimedocs Documentation

Release 1.0.2

Junior Teudjio

Sep 03, 2018

Contents

CHAPTER 1

What?

runtimedocs is a Python library that offers a sensible, customizable, human-friendly way to get a sense of what really happens during a function call. It implements a decorator which wraps your function/class and prints its runtime behavior and also saves it to a log file usually named as follows: module_name.function_name.rundimedocs.log.

CHAPTER 2

Why?

If you ever found yourself in one or more of these situations then runtimedocs could really help:

- you would like to know which function called your function(the one you decorated using runtimedocs).
- you would like to know what were the positional and key-word arguments received by your function at runtime.
- you want to write docstrings for a (legacy)function with unclear parameters naming and would like to know more about them to help you get started. For instance if that function expect a parameter named “x”, you are much more advanced if you know that during runtime “x” is usually of type=list, of value=[‘bar’, ‘foo’], of len=2.
- likewise, you may want to debug a function but ignore what are its expected input parameters and returned values (ie: their types and values). A good idea could be to decorate that function in your running environment and runtimedocs will log both the successfull and failling calls along with their inputs(types, and values) so you can know which paramters actually break your functiom.
- you would like to know if a function is multi output(eg: return foo, bla) or single output(eg: return foo). along with the types and values of the returned variables.
- your function runs well on a given host but breaks on another, runtimedocs tells you the hostname of the computer running your function.

CHAPTER 3

Features

- what was the expected signature of a function/class.
- what was the actual signature used when calling that function/class at run time.
- where was the function/class declared in and called from.
- what's the hostname of the machine running the code.
- what are the types, names, values of the input parameters and returned values of that function.
- when relevant also add specific information like their : len, signature, inheritance_tree, etc ...
- what were the positional/key-word arguments at call time.
- has the function exited successfully or ran into an exception.
- how long it took to run in case of success.
- display what was the exception otherwise and raises it back to not side-effect your program.

All these information are saved in a file usually named as follows: module_name.function_name.rundimedocs.log Additionally they can be printed on the terminal if the verbosity level is set to 1. You can easily toggle the runtimedocs decorator off by setting the env variable DISABLE_RUNTIMEDOCS to True.

CHAPTER 4

Quickstart

```
$ pip install runtimedocs
```

```
>>> from runtimedocs import runtimedocs

>>> #decorate the function/class of your choice
>>> @runtimedocs(verbosity=1, timing_info=False) #verbosity=1 means also print the␣
↪logs on terminal. timing_info=False means don't log time.
... def myadd(a, b, f=sum, not_used=None):
...     return f([a, b])
...
>>> @runtimedocs(verbosity=1, timing_info=False)
... def mysum(elements):
...     return sum(elements)
...
>>> #call the decorated function and see the runtime documentation printed on the␣
↪terminal and saved to a file called: __main__.myadd.runtimedocs.log
>>> myadd(1, 2, f=sum)
################################################################################
↪###############
#calling [myadd] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#-------------------------------------------------------------------------------
↪---------------
#declared signature = myadd(a, b, f=<built-in function sum>, not_used=None)
#called   signature = myadd(<class 'int'>, <class 'int'>, f=<class 'builtin_function_
↪or_method'>)
#-------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 2
#    #0:
#     type = <class 'int'>
#     value = 1
#-----
```

(continues on next page)

(continued from previous page)

```
#    #1:
#     type = <class 'int'>
#     value = 2
#-----
#Number of key word paramters: 1
#    f:
#     type = <class 'builtin_function_or_method'>
#     name = sum
#     signature = (iterable, start=0, /)
#     fullargspec = FullArgSpec(args=['iterable', 'start'], varargs=None, varkw=None,␣
↪defaults=None, kwonlyargs=[], kwonlydefaults=None, annotations={})
#     isbuiltin = True
#-----
#-------------------------------------------------------------------------------------
↪---------------
#[myadd] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----

>>> mysum([1, 2]) #logs printed and saved to a file called: __main__.mysum.
↪runtimedocs.log
#####################################################################################
↪###############
#calling [mysum] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#-------------------------------------------------------------------------------------
↪---------------
#declared signature = mysum(elements)
#called   signature = mysum(<class 'list'>)
#-------------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 1
#    #0:
#     type = <class 'list'>
#     len = 2
#     value = [1, 2]
#-----
#Number of key word paramters: 0
#-------------------------------------------------------------------------------------
↪---------------
#[mysum] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----

>>> mysum(el for el in [1,2])
#####################################################################################
↪################
#calling [mysum] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#-------------------------------------------------------------------------------------
↪---------------
```

(continues on next page)

(continued from previous page)

```
#declared signature = mysum(elements)
#called   signature = mysum(<class 'generator'>)
#-------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 1
#    #0:
#     type = <class 'generator'>
#     value = <generator object <genexpr> at 0x107b664f8>
#-----
#Number of key word paramters: 0
#-------------------------------------------------------------------------------
↪---------------
#[mysum] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----
```

CHAPTER 5

User's Guide

5.1 Disabling runtimedocs

Disable runtimedocs globally:

```
>>> import os
>>> #set the DISABLE_RUNTIMEDOCS to '1' which will casted to True (like any other non-
↪empty string).
>>> os.environ['DISABLE_RUNTIMEDOCS'] = '1'
>>> #with DISABLE_RUNTIMEDOCS env variable set to True, runtimedocs decorator doesn't␣
↪wrap your function, so calling these functions wont't print or save any log file.
>>> myadd(1, 2)
>>> mysum([1, 2])
```

Disable runtimedocs globally but force enable locally:

```
>>> import os
>>> #set the DISABLE_RUNTIMEDOCS to '1' which will casted to True (like any other non-
↪empty string).
>>> os.environ['DISABLE_RUNTIMEDOCS'] = '1'
>>> @runtimedocs(verbosity=1, timing_info=False, force_enable_runtimedocs=True)
... def mysum(elements):
...     return sum(elements)
...
>>> myadd(1, 2) #no logs for myadd
>>> mysum([1, 2]) #force_enable_runtimedocs is set to True for mysum so runtimedocs␣
↪will log the function call.
################################################################################
↪################
#calling [mysum] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#------------------------------------------------------------------------------
↪---------------
```

(continues on next page)

(continued from previous page)

```
#declared signature = mysum(elements)
#called   signature = mysum(<class 'list'>)
#-------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 1
#    #0:
#     type = <class 'list'>
#     len = 2
#     value = [1, 2]
#-----
#Number of key word paramters: 0
#-------------------------------------------------------------------------------
↪---------------
#[mysum] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----
```

5.2 Customizations

Customized how runtimedocs parse a given type:

```
>>> from collections import OrderedDict
>>> # define the function to parse a type as you like, preferably return an orderdict␣
↪to see them printed in the order you want.
>>> def my_custom_list_parser_func(L):
...     return OrderedDict(
...         bar = 'bar',
...         foo = 'foo',
...         mylist_type = type(L),
...         mylist_len = len(L),
...         mylist_repr =repr(L))
...
>>> custom_parsers_dict = {"<class 'list'>": my_custom_list_parser_func}
>>> @runtimedocs(verbosity=1, timing_info=False, custom_types_parsers_dict=custom_
↪parsers_dict)
... def mysum(elements):
...     return sum(elements)
...
>>> mysum([1,2])
################################################################################
↪###############
#calling [mysum] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#-------------------------------------------------------------------------------
↪---------------
#declared signature = mysum(elements)
#called   signature = mysum(<class 'list'>)
#-------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 1
#    #0:
```

(continues on next page)

(continued from previous page)

```
#     bar = bar
#     foo = foo
#     mylist_type = <class 'list'>
#     mylist_len = 2
#     mylist_repr = [1, 2]
#-----
#Number of key word paramters: 0
#-------------------------------------------------------------------------------------
↪---------------
#[mysum] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----
```

Aggregate all the logs for multiple functions in a same file:

```
>>> import logging
>>> file_handler = logging.FileHandler('aggregation.runtimedocs.log')

>>> @runtimedocs(extra_logger_handlers=[file_handler])
>>> def myadd(a, b, f=sum, not_used=None):
...     return f([a, b])
...
>>> #even faster, you can also directly pass the string as an extra_hanlder
>>> @runtimedocs(extra_logger_handlers=['aggregation.runtimedocs.log'])
>>> def mysum(elements):
...     return sum(elements)
...
>>> # after running these two functions 3 log files will be created: 2 for each␣
↪function as usual and a 3rd one for the agregated logs
>>> mysum([1,2])
>>> myadd(1, 2, f=sum)
>>> # content of aggregation.runtimedocs.log :
#######################################################################################
↪###############
#calling [myadd] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#-------------------------------------------------------------------------------------
↪---------------
#declared signature = myadd(a, b, f=<built-in function sum>, not_used=None)
#called   signature = myadd(<class 'int'>, <class 'int'>, f=<class 'builtin_function_
↪or_method'>)
#-------------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 2
#    #0:
#     type = <class 'int'>
#     value = 1
#-----
#    #1:
#     type = <class 'int'>
#     value = 2
#-----
#Number of key word paramters: 1
#    f:
```

(continues on next page)

(continued from previous page)

```
#     type = <class 'builtin_function_or_method'>
#     name = sum
#     signature = (iterable, start=0, /)
#     fullargspec = FullArgSpec(args=['iterable', 'start'], varargs=None, varkw=None,␣
↪defaults=None, kwonlyargs=[], kwonlydefaults=None, annotations={})
#     isbuiltin = True
#-----
#-------------------------------------------------------------------------------------
↪---------------
#[myadd] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----
#####################################################################################
↪################
#calling [mysum] declared inside module [__main__]
#caller name: [runtimedocs.core]
#ran inside: hostname=[Juniors-MBP.lan]
#-------------------------------------------------------------------------------------
↪---------------
#declared signature = mysum(elements)
#called   signature = mysum(<class 'list'>)
#-------------------------------------------------------------------------------------
↪---------------
#Number of positional paramters: 1
#    #0:
#     type = <class 'list'>
#     len = 2
#     value = [1, 2]
#-----
#Number of key word paramters: 0
#-------------------------------------------------------------------------------------
↪---------------
#[mysum] ran successfully in [0.0]seconds and its returned value has these specs:
#single output return statement:
#     type = <class 'int'>
#     value = 3
#-----
```

CHAPTER 6

API Guide

6.1 runtimedocs

6.1.1 runtimedocs package

6.1.1.1 Submodules

6.1.1.2 runtimedocs.core module

runtimedocs.core.**runtimedocs**(*force_enable_runtimedocs=False*, *verbosity=0*, *timing_info=True*, *default_type_parser=<function default_type_parser>*, *max_stringify=1000*, *prefix_module_name_to_logger_name=True*, *custom_logger_name=None*, *extra_logger_handlers=None*, *common_types_parsers_dict=ChainMap({}, {"<class 'function'>": <function function_parser>, "<class 'type'>": <function class_parser>, "<class 'builtin_function_or_method'>": <function function_parser>})*, *custom_types_parsers_dict=None*)

runtimedocs decorator helps you understand how your code behaves at runtime.

It provides detailed information about: - what was the expected signature of a function/class. - what was the actual signature used when calling that function/class at run time. - where was the function/class declared in and called from. - what's the hostname of the machine running the code. - what are the types, names, values of the input parameters and returned values of that function. - when relevant also add specific information like their : len, signature, inheritance_tree, etc ... - what were the positional/key-word arguments at call time. - has the function exited successfully or ran into an exception. - how long it took to run in case of success. - display what was the exception otherwise and raises it back to not side-effect your program.

All these information are saved in a file usually named as follows: module_name.function_name.rundimedocs.log Additionally they can be printed on the terminal if the verbosity level is set to 1. You can easily toggle the runtimedocs decorator off by setting the env variable DISABLE_RUNTIMEDOCS to True.

Parameters

- **force_enable_runtimedocs** (*bool | DEFAULT = False*) – In case the environment variable DISABLE_RUNTIMEDOCS is set to True, setting this flag to True allows you to activate the decorator for a specific function/class.
- **verbosity** (*int | DEFAULT = 0*) – When set to 0, it means the runtimedocs information won't be printed on the terminal This allows you to still see your usual printed messages easily. When set to a value > 0, it means rundimedocs will also print on the terminal what its been saved in the runtimedocs log file of the decorated function.
- **timing_info** (*bool | DEFAULT = True*) – True, means you want to keep track and log the time at which the decorated function was called
- **default_type_parser** (*function | DEFAULT = runtimedocs.helpers.default_type_parser*) – default way to parse the input parameters and returned values. This will take as input for instance one the arguments been passed in to the decorated function and return an OrderDict with keys: type, value. But also len and keys when relevant.
- **max_stringify** (*int | DEFAULT = 1000*) – this value is used by the default_type_parser function to chunk the length of the string returned by repr of the arg been parsed. ie: value_of_arg_been_parsed = repr(arg_been_parsed)[:max_stringify]
- **prefix_module_name_to_logger_name** (*bool | DEFAULT = True*) – True, means that runtimedocs decorator will save the information for a specific function/class been decorated in a file called: current_module_name.decorated_function_name.rundimedocs.log if False that file is called: decorated_function_name.rundimedocs.log
- **custom_logger_name** (*str | DEFAULT = None*) – if a string is specified, this will be the name of the logger and runtimedocs will save information in a file called: custom_logger_name.runtimedocs.log no matter what's the value of prefix_module_name_to_logger_name
- **extra_logger_handlers** (*list | DEFAULT = None*) – runtimedocs decorator uses the builting logging module to create log information. So this argument allows you to specify additional [file]handlers to where to save the runtime information being extracted. This could be useful for example to centralized all the logged info in a single file or group of files since by default every function in every module has its own log file. Note that, this allows you to add additional handlers, not overrides the default one. Also each handler of the list could be a string or a custom instance of logging.FileHandler()
- **common_types_parsers_dict** (*dict | DEFAULT = helpers.common_types_parsers_dict*) – this parameter allows you to bypass the default_type_parser for certain specific builtin python types it is a dictionary with keys representing the type as str and the parsing functions as values. If the runtimedocs_types_parsers plugin is installed then additional parsers for third-parties types are available and will bypass the default_type_parser. For instance if the plugin is installed, new parsers are avalaible for numpy, scipy, pandas enabling runtimedocs to print even more relevant information like: shape, dim, mean, std, etc ...
- **custom_types_parsers_dict** (*dict | DEFAULT = None*) – similarly to common_types_parsers_dict but for your own custom types. For instance if your program makes uses of a objects from a class you want to parse in a given way then do: custom_types_parsers_dict = {"<class 'MyClassName'>" : my_class_parser_func } where my_class_parser_func returns an OrderDict with keys like: type, value, my_size, etc... Another use of it, is if you want to parse nested lists, the default_type_parser can do that but by overriding the parsing function for the type: "<class'list'>" you have more control on how you want to parse the nested lists.

Returns **wrapper** – the decorated function/class.

Return type function

6.1.1.3 runtimedocs.helpers module

runtimedocs.helpers.**caller_name**(*skip=2*)
Get a name of a caller in the format module.class.method

skip specifies how many levels of stack to skip while getting caller name. skip=1 means "who calls me", skip=2 "who calls my caller" etc.

An empty string is returned if skipped levels exceed stack height

copied from here: https://stackoverflow.com/questions/2654113/python-how-to-get-the-callers-method-name-in-the-called-meth

runtimedocs.helpers.**class_parser**(*arg*)
type parser for user defined and builtin classes. :Parameters: **arg** (*class to parse*)

Returns **parsed**

Return type OrderedDict('value', 'signature', 'fullargspec', 'isbuiltin', 'inheritance_tree)

runtimedocs.helpers.**default_type_parser**(*arg*, *max_stringify=1000*)
default type parser which basically return the repr string of the object.

Parameters

- **arg** (*object to parse*)
- **max_stringify** (*how long at max should be the returned string after doing repr(arg)*)

Returns **parsed** – value: is repr(arg)[:max_stringify] keys: is the the keys in the parsed object and only added to parsed if the object is a dict len: is the the length of the parsed object and only added to parsed if the object is an iterable

Return type OrderedDict('value', ['keys'], ['len'])

runtimedocs.helpers.**function_parser**(*arg*)
type parser for user defined and builtin functions.

Parameters **arg** (*function to parse*)

Returns **parsed**

Return type OrderedDict('value', 'signature', 'fullargspec', 'isbuiltin')

runtimedocs.helpers.**get_type**(*arg*)
helper function the get the type of an abject as a string.

6.1.1.4 Module contents

6.2 runtimedocs.core module

runtimedocs.core.**runtimedocs**(*force_enable_runtimedocs=False*, *verbosity=0*, *timing_info=True*, *default_type_parser=<function default_type_parser>*, *max_stringify=1000*, *prefix_module_name_to_logger_name=True*, *custom_logger_name=None*, *extra_logger_handlers=None*, *common_types_parsers_dict=ChainMap({}, {"<class 'function'>": <function function_parser>, "<class 'type'>": <function class_parser>, "<class 'builtin_function_or_method'>": <function function_parser>})*, *custom_types_parsers_dict=None*)

runtimedocs decorator helps you understand how your code behaves at runtime.

It provides detailed information about: - what was the expected signature of a function/class. - what was the actual signature used when calling that function/class at run time. - where was the function/class declared in and called from. - what's the hostname of the machine running the code. - what are the types, names, values of the input parameters and returned values of that function. - when relevant also add specific information like their : len, signature, inheritance_tree, etc ... - what were the positional/key-word arguments at call time. - has the function exited successfully or ran into an exception. - how long it took to run in case of success. - display what was the exception otherwise and raises it back to not side-effect your program.

All these information are saved in a file usually named as follows: module_name.function_name.rundimedocs.log Additionally they can be printed on the terminal if the verbosity level is set to 1. You can easily toggle the runtimedocs decorator off by setting the env variable DISABLE_RUNTIMEDOCS to True.

Parameters

- **force_enable_runtimedocs** (*bool | DEFAULT = False*) – In case the environment variable DISABLE_RUNTIMEDOCS is set to True, setting this flag to True allows you to activate the decorator for a specific function/class.
- **verbosity** (*int | DEFAULT = 0*) – When set to 0, it means the runtimedocs information won't be printed on the terminal This allows you to still see your usual printed messages easily. When set to a value > 0, it means rundimedocs will also print on the terminal what its been saved in the runtimedocs log file of the decorated function.
- **timing_info** (*bool | DEFAULT = True*) – True, means you want to keep track and log the time at which the decorated function was called
- **default_type_parser** (*function | DEFAULT = runtimedocs.helpers.default_type_parser*) – default way to parse the input parameters and returned values. This will take as input for instance one the arguments been passed in to the decorated function and return an OrderDict with keys: type, value. But also len and keys when relevant.
- **max_stringify** (*int | DEFAULT = 1000*) – this value is used by the default_type_parser function to chunk the length of the string returned by repr of the arg been parsed. ie: value_of_arg_been_parsed = repr(arg_been_parsed)[:max_stringify]
- **prefix_module_name_to_logger_name** (*bool | DEFAULT = True*) – True, means that runtimedocs decorator will save the information for a specific function/class been decorated in a file called: current_module_name.decorated_function_name.rundimedocs.log if False that file is called: decorated_function_name.rundimedocs.log
- **custom_logger_name** (*str | DEFAULT = None*) – if a string is specified, this will be the name of the logger and runtimedocs will save information in a file called: custom_logger_name.runtimedocs.log no matter what's the value of prefix_module_name_to_logger_name
- **extra_logger_handlers** (*list | DEFAULT = None*) – runtimedocs decorator uses the builting logging module to create log information. So this argument allows you to specify additional [file]handlers to where to save the runtime information being extracted. This could be useful for example to centralized all the logged info in a single file or group of files since by default every function in every module has its own log file. Note that, this allows you to add additional handlers, not overrides the default one. Also each handler of the list could be a string or a custom instance of logging.FileHandler()
- **common_types_parsers_dict** (*dict | DEFAULT = helpers.common_types_parsers_dict*) – this parameter allows you to bypass the default_type_parser for certain specific builtin

python types it is a dictionary with keys representing the type as str and the parsing functions as values. If the runtimedocs_types_parsers plugin is installed then additional parsers for third-parties types are available and will bypass the default_type_parser. For instance if the plugin is installed, new parsers are avalaible for numpy, scipy, pandas enabling runtimedocs to print even more relevant information like: shape, dim, mean, std, etc ...

- **custom_types_parsers_dict** (*dict | DEFAULT = None*) – similarly to common_types_parsers_dict but for your own custom types. For instance if your program makes uses of a objects from a class you want to parse in a given way then do: custom_types_parsers_dict = {"<class 'MyClassName'>" : my_class_parser_func } where my_class_parser_func returns an OrderDict with keys like: type, value, my_size, etc... Another use of it, is if you want to parse nested lists, the default_type_parser can do that but by overriding the parsing function for the type: "<class'list'>" you have more control on how you want to parse the nested lists.

Returns **wrapper** – the decorated function/class.

Return type function

6.3 runtimedocs.helpers module

runtimedocs.helpers.**caller_name**(*skip=2*)
Get a name of a caller in the format module.class.method

skip specifies how many levels of stack to skip while getting caller name. skip=1 means "who calls me", skip=2 "who calls my caller" etc.

An empty string is returned if skipped levels exceed stack height

copied from here: https://stackoverflow.com/questions/2654113/python-how-to-get-the-callers-method-name-in-the-called-meth

runtimedocs.helpers.**class_parser**(*arg*)
type parser for user defined and builtin classes. :Parameters: **arg** (*class to parse*)

Returns **parsed**

Return type OrderedDict('value', 'signature', 'fullargspec', 'isbuiltin', 'inheritance_tree)

runtimedocs.helpers.**default_type_parser**(*arg*, *max_stringify=1000*)
default type parser which basically return the repr string of the object.

Parameters

- **arg** (*object to parse*)
- **max_stringify** (*how long at max should be the returned string after doing repr(arg)*)

Returns **parsed** – value: is repr(arg)[:max_stringify] keys: is the the keys in the parsed object and only added to parsed if the object is a dict len: is the the length of the parsed object and only added to parsed if the object is an iterable

Return type OrderedDict('value', ['keys'], ['len'])

runtimedocs.helpers.**function_parser**(*arg*)
type parser for user defined and builtin functions.

Parameters **arg** (*function to parse*)

Returns **parsed**

Return type OrderedDict('value', 'signature', 'fullargspec', 'isbuiltin')

runtimedocs.helpers.**get_type**(*arg*)
helper function the get the type of an abject as a string.

6.4 Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index