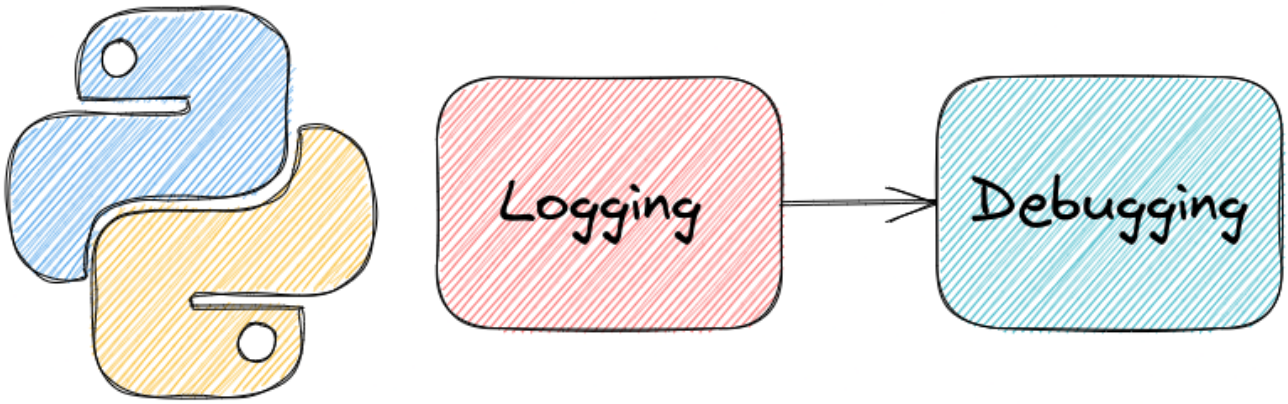
Efficient Python Tricks and Tools for Data Scientists - By Khuyen Tran

Logging and Debugging

GitHub View on GitHub Book View Book

Collections of tools for logging and debugging Python code.



rich.inspect: Produce a Beautiful Report on any Python Object

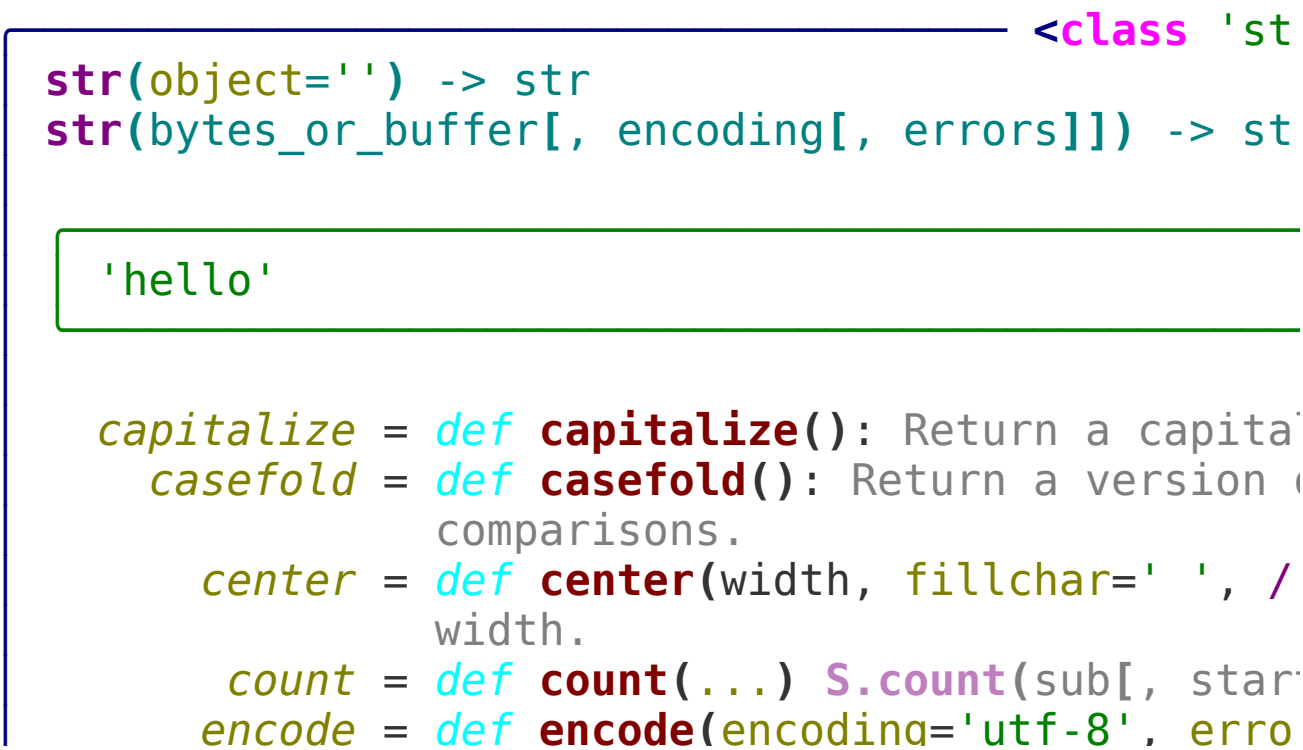
```
$ pip install rich
```

If you want to quickly see which attributes and methods of a Python object are available, use rich's `inspect` method.

rich's `inspect` method allows you to create a beautiful report for any Python object, including a string.

```
from rich import inspect

print(inspect('hello', methods=True))
```



```
                codec registered for encoding.
  endswith = def endswith(...) S.endswith(suff
expandtabs = def expandtabs(tabsize=8): Return
                expanded using spaces.
      find = def find(...) S.find(sub[, start[
    format = def format(...) S.format(*args, *
format_map = def format_map(...) S.format_map(
     index = def index(...) S.index(sub[, star
   isalnum = def isalnum(): Return True if the
                otherwise.
   isalpha = def isalpha(): Return True if the
                otherwise.
   isascii = def isascii(): Return True if all
                False otherwise.

 isdecimal = def isdecimal(): Return True if t
                otherwise.
   isdigit = def isdigit(): Return True if the
                otherwise.
isidentifier = def isidentifier(): Return True i
                identifier, False otherwise.
   islower = def islower(): Return True if the
                otherwise.
 isnumeric = def isnumeric(): Return True if t
                otherwise.
isprintable = def isprintable(): Return True if
                otherwise.
   isspace = def isspace(): Return True if the
                otherwise.
   istitle = def istitle(): Return True if the
                otherwise.
   isupper = def isupper(): Return True if the
                otherwise.
      join = def join(iterable, /): Concatenat
     ljust = def ljust(width, fillchar=' ', /)
                length width.
     lower = def lower(): Return a copy of the
     lstrip = def lstrip(chars=None, /): Return
                whitespace removed.
 maketrans = def maketrans(...) Return a trans
 partition = def partition(sep, /): Partition
                given separator.
```

```
    replace = def replace(old, new, count=-1, /
              substring old replaced by new.
     rfind = def rfind(...) S.rfind(sub[, star
    rindex = def rindex(...) S.rindex(sub[, sta
     rjust = def rjust(width, fillchar=' ', /)
              length width.
rpartition = def rpartition(sep, /): Partition
              given separator.
    rsplit = def rsplit(sep=None, maxsplit=-1)
              string, using sep as the delimite
    rstrip = def rstrip(chars=None, /): Return
              whitespace removed.
     split = def split(sep=None, maxsplit=-1):
              string, using sep as the delimite
```

```
splitlines = def splitlines(keepends=False): R
              breaking at line boundaries.
startswith = def startswith(...) S.startswith(
     strip = def strip(chars=None, /): Return a
              trailing whitespace removed.
  swapcase = def swapcase(): Convert uppercase
              characters to uppercase.
     title = def title(): Return a version of t
 translate = def translate(table, /): Replace c
              given translation table.
     upper = def upper(): Return a copy of the
     zfill = def zfill(width, /): Pad a numeric
              a field of the given width.
```

Rich's Console: Debug your Python Function in One Line of Code

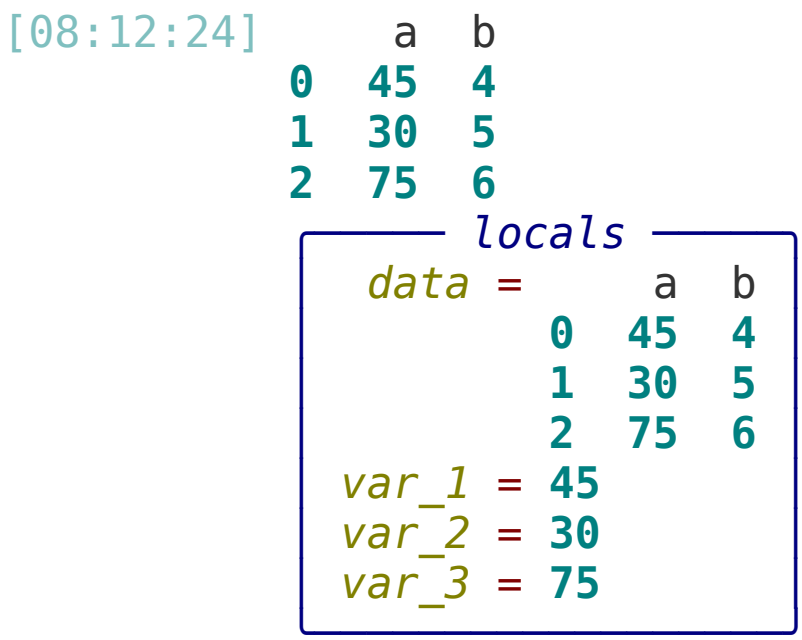
```
$ pip install rich
```

Sometimes, you might want to know which elements in the function created a certain output. Instead of printing every variable in the function, you can simply use Rich's `Console` object to print both the output and all the variables in the function.

```
from rich import console
from rich.console import Console
import pandas as pd

console = Console()
data = pd.DataFrame({'a': [1, 2, 3], 'b': [4,
5, 6]})

def edit_data(data):
    var_1 = 45
    var_2 = 30
    var_3 = var_1 + var_2
    data['a'] = [var_1, var_2, var_3]
    console.log(data, log_locals=True)
edit_data(data)
```



Link to my article about rich.

Link to rich.

loguru: Print Readable Traceback in Python

```
$ pip install loguru
```

Sometimes, it is difficult to understand the traceback and to know which inputs cause the error. Is there a way that you can print a more readable traceback?

That is when loguru comes in handy. By adding decorator `logger.catch` to a function, loguru logger will print a more readable trackback and save the traceback to a separate file like below

```python
from sklearn.metrics import mean_squared_error
import numpy as np
from loguru import logger

logger.add("file_{time}.log", format="{time}
{level} {message}")

@logger.catch
def evaluate_result(y_true: np.array, y_pred:
np.array):
    mean_square_err =
mean_squared_error(y_true, y_pred)
    root_mean_square_err = mean_square_err **
0.5
    y_true = np.array([1, 2, 3])

y_pred = np.array([1.5, 2.2])
evaluate_result(y_true, y_pred)
```

```
> File "/tmp/ipykernel_174022/1865479429.py",
line 14, in <module>
    evaluate_result(y_true, y_pred)
    |               |       └ array([1.5,
2.2])
    |               └ array([1, 2, 3])
    └ <function evaluate_result at
0x7f279588f430>

  File "/tmp/ipykernel_174022/1865479429.py",
line 9, in evaluate_result
    mean_square_err =
mean_squared_error(y_true, y_pred)
                       |                  |
  └ array([1.5, 2.2])
                       |                  └
array([1, 2, 3])
                       └ <function
mean_squared_error at 0x7f27958bfca0>
```

```
 File
"/home/khuyen/book/venv/lib/python3.8/site-
packages/sklearn/utils/validation.py", line
63, in inner_f
    return f(*args, **kwargs)
           |  |       └ {}
           |  └ (array([1, 2, 3]), array([1.5,
2.2]))
           └ <function mean_squared_error at
0x7f27958bfb80>
  File
"/home/khuyen/book/venv/lib/python3.8/site-
packages/sklearn/metrics/_regression.py", line
335, in mean_squared_error
    y_type, y_true, y_pred, multioutput =
_check_reg_targets(
           |       |                      └
<function _check_reg_targets at
0x7f27958b7af0>
           |       └ array([1.5, 2.2])
           └ array([1, 2, 3])
```

```
  File
"/home/khuyen/book/venv/lib/python3.8/site-
packages/sklearn/metrics/_regression.py", line
88, in _check_reg_targets
    check_consistent_length(y_true, y_pred)
    |                       |       └
array([1.5, 2.2])
    |                       └ array([1, 2, 3])
    └ <function check_consistent_length at
0x7f279676e040>
  File
"/home/khuyen/book/venv/lib/python3.8/site-
packages/sklearn/utils/validation.py", line
319, in check_consistent_length
    raise ValueError("Found input variables
with inconsistent numbers of"

ValueError: Found input variables with
inconsistent numbers of samples: [3, 2]
```

Link to loguru.

Icrecream: Never use print() to debug again

```
$ pip install icecream
```

If you use print or log to debug your code, you might be confused about which line of code creates the output, especially when there are many outputs.

You might insert text to make it less confusing, but it is time-consuming.

```
from icecream import ic

def plus_one(num):
    return num + 1

print('output of plus_on with num = 1:',
plus_one(1))
print('output of plus_on with num = 2:',
plus_one(2))
```

```
output of plus_on with num = 1: 2
output of plus_on with num = 2: 3
```

Try icecream instead. Icrecream inspects itself and prints both its own arguments and the values of those arguments like below.

```
ic(plus_one(1))
ic(plus_one(2))
```

```
ic| plus_one(1): 2
ic| plus_one(2): 3

3
```

Output:

```
ic| plus_one(1): 2
ic| plus_one(2): 3
```

Link to icecream

Link to my article about icecream

heartrate — Visualize the Execution of a Python Program in Real-Time

```
$ pip install heartrate
```

If you want to visualize which lines are executed and how many times they are executed, try heartrate.

You only need to add two lines of code to use heartrate.

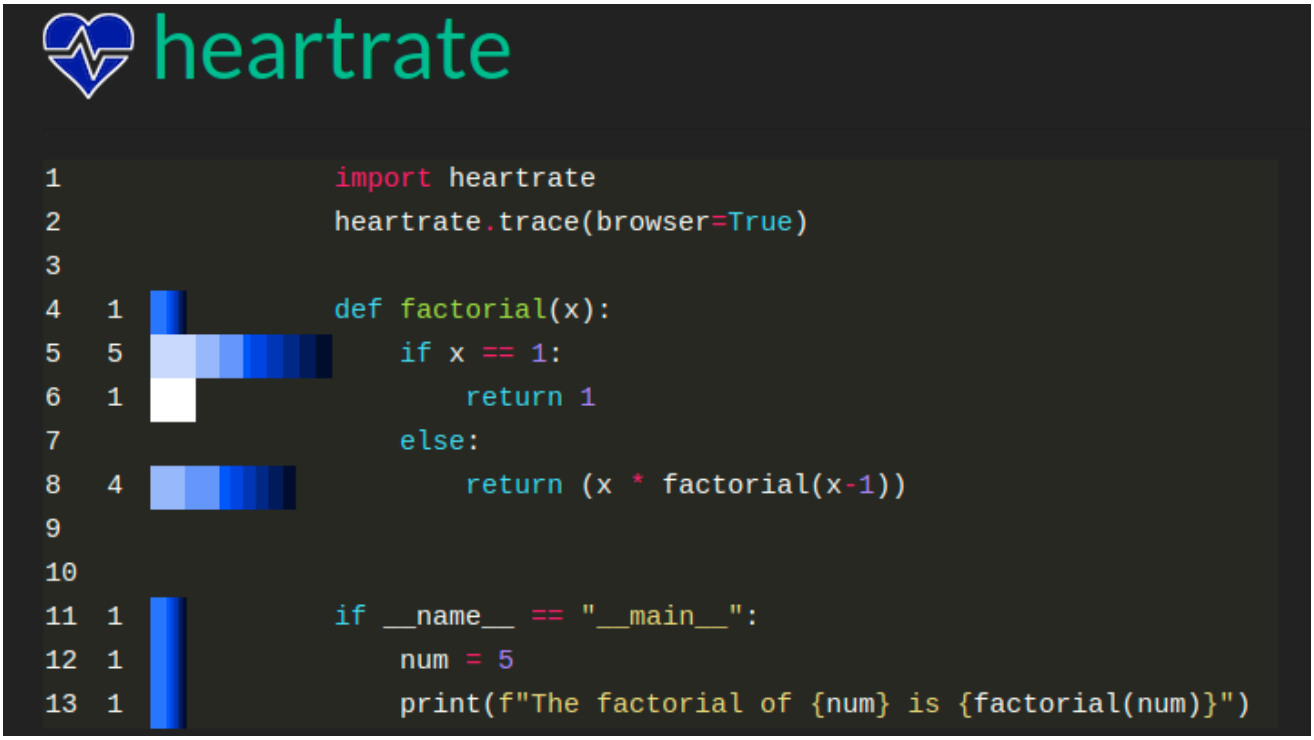
```
import heartrate
heartrate.trace(browser=True)

def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


if __name__ == "__main__":
    num = 5
    print(f"The factorial of {num} is
{factorial(num)}")
```

```
 * Serving Flask app 'heartrate.core' (lazy
loading)
 * Environment: production
The factorial of 5 is 120
Opening in existing browser session.
```

You should see something similar to the below when opening the browser:



Link to heartrate.

snoop : Smart Print to Debug your Python Function

```
$ pip install snoop
```

If you want to figure out what is happening in your code without adding many print statements, try snoop.

To use snoop, simply add the `@snoop` decorator to a function you want to understand.

```
import snoop

@snoop
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


if __name__ == "__main__":
    num = 5
    print(f"The factorial of {num} is
{factorial(num)}")
```

```
10:19:00.73 >>> Call to factorial in File "
<ipython-input-2-57aff36d5f6d>", line 4
10:19:00.73 ...... x = 5
10:19:00.73    4 | def factorial(x):
10:19:00.73    5 |     if x == 1:
10:19:00.73    8 |         return (x *
factorial(x-1))
    10:19:00.74 >>> Call to factorial in File
"<ipython-input-2-57aff36d5f6d>", line 4
    10:19:00.74 ...... x = 4
    10:19:00.74    4 | def factorial(x):
    10:19:00.74    5 |     if x == 1:
    10:19:00.74    8 |         return (x *
factorial(x-1))
        10:19:00.74 >>> Call to factorial in
File "<ipython-input-2-57aff36d5f6d>", line 4
        10:19:00.74 ...... x = 3
        10:19:00.74    4 | def factorial(x):
        10:19:00.74    5 |     if x == 1:
        10:19:00.75    8 |         return (x *
factorial(x-1))
            10:19:00.75 >>> Call to factorial
in File "<ipython-input-2-57aff36d5f6d>", line
4
            10:19:00.75 ...... x = 2
            10:19:00.75    4 | def
factorial(x):
            10:19:00.75    5 |     if x == 1:
            10:19:00.75    8 |         return
(x * factorial(x-1))
```

```
                10:19:00.75 >>> Call to
factorial in File "<ipython-input-2-
57aff36d5f6d>", line 4
                10:19:00.75 ...... x = 1
                10:19:00.75    4 | def
factorial(x):
                10:19:00.76    5 |     if x ==
1:
                10:19:00.76    6 |
return 1
                10:19:00.76 <<< Return value
from factorial: 1
            10:19:00.76    8 |         return
(x * factorial(x-1))
            10:19:00.77 <<< Return value from
factorial: 2
        10:19:00.77    8 |         return (x *
factorial(x-1))
        10:19:00.77 <<< Return value from
factorial: 6
    10:19:00.77    8 |         return (x *
factorial(x-1))
    10:19:00.77 <<< Return value from
factorial: 24
10:19:00.78    8 |         return (x *
factorial(x-1))
10:19:00.78 <<< Return value from factorial:
120
```

```
The factorial of 5 is 120
```

Logging in Pandas Pipelines

```
$ pip install scikit-lego
```

When using pandas pipe, you might want to check whether each pipeline transforms your pandas DataFrame correctly. To automatically log the information of a pandas DataFrame after each pipeline, use the decorator `sklego.pandas_utils.log_step`.

```
import pandas as pd
from sklego.pandas_utils import log_step
import logging
```

```
df = pd.DataFrame({"col1": [1, 2, 3], "col2":
["a", "b", "c"]})
```

To use `log_step`, simply use it as a decorator for functions being applied to your DataFrame.

```
@log_step(print_fn=logging.info)
def make_copy(df: pd.DataFrame):
    return df.copy()
```

```
@log_step(print_fn=logging.info)
def drop_column(df: pd.DataFrame):
    return df[["col2"]]


@log_step(print_fn=logging.info)
def encode_cat_variables(df: pd.DataFrame):
    df["col2"] = df["col2"].map({"a": 1, "b":
2, "c": 3})
    return df
```

```
df =
df.pipe(make_copy).pipe(drop_column).pipe(enco
de_cat_variables)
```

```
INFO:root:[make_copy(df)] time=0:00:00.000239
n_obs=3, n_col=2
INFO:root:[drop_column(df)]
time=0:00:00.002117 n_obs=3, n_col=1
INFO:root:[encode_cat_variables(df)]
time=0:00:00.003217 n_obs=3, n_col=1
```

Find more ways to customize your logging here.

Add Progress Bar to Your List Comprehension

```
$ pip install tqdm
```

If your for loop or list comprehension takes a long time to run, you might want to know which element is being processed. You can add clarity to your for-loop by using tqdm. Using tqdm with an iterable will show a progress bar.

```
from tqdm.notebook import tqdm
from time import sleep


def lower(word):
    sleep(1)
    print(f"Processing {word}")
    return word.lower()


words = tqdm(["Duck", "dog", "Flower", "fan"])

[lower(word) for word in words]
```

```
  0%|          | 0/4 [00:00<?, ?it/s]
```

```
Processing Duck
Processing dog
Processing Flower
Processing fan
```

```
['duck', 'dog', 'flower', 'fan']
```

Link to tqdm.