

Watch by Tuesday, December 1, 2020 | **Lesson #16**

Functions, organization, external packages, and string formatting

OCEAN 215 | Autumn 2020
Ethan Campbell and **Katy Christensen**

What we'll cover in this lesson

1. Creating functions
2. Organizing your code and installing external packages
3. String formatting

What we'll cover in this lesson

1. **Creating functions**
2. Organizing your code and installing external packages
3. String formatting

What we'll cover in this lesson

1. Creating functions
- 2. Organizing your code and installing external packages**
3. String formatting

Code organization

**You want your code to be easily read and debugged.
It is often best to “tell a story” with your code...**

Put the code in a logical order with intuitive blocks.

Each cell in Colab should have a specific task.

Use comments to denote what is happening, but not too many!

Put functions early in the code in the order that you use them later.

```
[ ] 1 # # Run this code only once per notebook, then comment it out  
2 # !pip install netcdf4
```

```
[ ] 1 import numpy as np  
2 import xarray as xr  
3 import matplotlib.pyplot as plt  
4  
5 import sys  
6 sys.path.append('/content/drive/My Drive')  
7 import my_module  
8  
9 from google.colab import drive  
10 drive.mount('/content/drive')
```

```
[ ] 1 def func_to_read_the_data(*Arguments*):  
2   # Do something with arguments  
3   # Return something  
4   return  
5  
6  
7 def func_to_analyze_the_data(*Arguments*):  
8   # Do something with arguments  
9   # Return something  
10  return  
11  
12  
13 def func_to_plot_the_data(*Arguments*):  
14   # Do something with arguments  
15   # Return something  
16   return
```

```
[ ] 1 filepath =  
2 filename =  
3  
4 # Read the data  
5 data = func_to_read_the_data()  
6  
7 # Any additional reading statements  
8
```

```
[ ] 1 # Analyze the data  
2 analysis = func_to_analyze_the_data()  
3  
4 # Any additional analysis  
5
```

```
[ ] 1 # Save/plot the data  
2 func_to_plot_the_data  
3
```

```
[ ] 1 # Any additional plotting
```

Set up your system and install any needed packages

```
[ ] 1 # # Run this code only once per notebook, then comment it out  
2 # !pip install netcdf4
```

```
[ ] 1 import numpy as np  
2 import xarray as xr  
3 import matplotlib.pyplot as plt  
4  
5 import sys  
6 sys.path.append('/content/drive/My Drive')  
7 import my_module  
8  
9 from google.colab import drive  
10 drive.mount('/content/drive')
```

```
[ ] 1 def func_to_read_the_data(*Arguments*):  
2   # Do something with arguments  
3   # Return something  
4   return  
5  
6  
7 def func_to_analyze_the_data(*Arguments*):  
8   # Do something with arguments  
9   # Return something  
10  return  
11  
12  
13 def func_to_plot_the_data(*Arguments*):  
14   # Do something with arguments  
15   # Return something  
16   return
```

```
[ ] 1 filepath =  
2 filename =  
3  
4 # Read the data  
5 data = func_to_read_the_data()  
6  
7 # Any additional reading statements  
8
```

```
[ ] 1 # Analyze the data  
2 analysis = func_to_analyze_the_data()  
3  
4 # Any additional analysis  
5
```

```
[ ] 1 # Save/plot the data  
2 func_to_plot_the_data  
3
```

```
[ ] 1 # Any additional plotting
```

Set up your system and install any needed packages

Import your packages/modules (including any custom modules)

Mount your drive

```
[ ] 1 # # Run this code only once per notebook, then comment it out  
2 # !pip install netcdf4
```

```
[ ] 1 import numpy as np  
2 import xarray as xr  
3 import matplotlib.pyplot as plt  
4  
5 import sys  
6 sys.path.append('/content/drive/My Drive')  
7 import my_module  
8  
9 from google.colab import drive  
10 drive.mount('/content/drive')
```

```
[ ] 1 def func_to_read_the_data(*Arguments*):  
2   # Do something with arguments  
3   # Return something  
4   return  
5  
6  
7 def func_to_analyze_the_data(*Arguments*):  
8   # Do something with arguments  
9   # Return something  
10  return  
11  
12  
13 def func_to_plot_the_data(*Arguments*):  
14   # Do something with arguments  
15   # Return something  
16   return
```

```
[ ] 1 filepath =  
2 filename =  
3  
4 # Read the data  
5 data = func_to_read_the_data()  
6  
7 # Any additional reading statements  
8
```

```
[ ] 1 # Analyze the data  
2 analysis = func_to_analyze_the_data()  
3  
4 # Any additional analysis  
5
```

```
[ ] 1 # Save/plot the data  
2 func_to_plot_the_data  
3
```

```
[ ] 1 # Any additional plotting
```

Set up your system and install any needed packages

Import your packages/modules (including any custom modules)

Mount your drive

Create different functions for frequently used code

It usually helps to create functions that do a specific tasks
(e.g. read, analyze, plot)

```
[ ] 1 # # Run this code only once per notebook, then comment it out
2 # !pip install netcdf4
```

```
[ ] 1 import numpy as np
2 import xarray as xr
3 import matplotlib.pyplot as plt
4
5 import sys
6 sys.path.append('/content/drive/My Drive')
7 import my_module
8
9 from google.colab import drive
10 drive.mount('/content/drive')
```

```
[ ] 1 def func_to_read_the_data(*Arguments*):
2   # Do something with arguments
3   # Return something
4   return
5
6
7 def func_to_analyze_the_data(*Arguments*):
8   # Do something with arguments
9   # Return something
10  return
11
12
13 def func_to_plot_the_data(*Arguments*):
14   # Do something with arguments
15   # Return something
16   return
```

```
[ ] 1 filepath =
2 filename =
3
4 # Read the data
5 data = func_to_read_the_data()
6
7 # Any additional reading statements
8
```

```
[ ] 1 # Analyze the data
2 analysis = func_to_analyze_the_data()
3
4 # Any additional analysis
5
```

```
[ ] 1 # Save/plot the data
2 func_to_plot_the_data
3
```

```
[ ] 1 # Any additional plotting
```

Set up your system and install any needed packages

Import your packages/modules (including any custom modules)

Mount your drive

Create different functions for frequently used code

It usually helps to create functions that do a specific tasks
(e.g. read, analyze, plot)

Read the data

Analyze the data

```
[ ] 1 # # Run this code only once per notebook, then comment it out
2 # !pip install netcdf4
```

```
[ ] 1 import numpy as np
2 import xarray as xr
3 import matplotlib.pyplot as plt
4
5 import sys
6 sys.path.append('/content/drive/My Drive')
7 import my_module
8
9 from google.colab import drive
10 drive.mount('/content/drive')
```

```
[ ] 1 def func_to_read_the_data(*Arguments*):
2   # Do something with arguments
3   # Return something
4   return
5
6
7 def func_to_analyze_the_data(*Arguments*):
8   # Do something with arguments
9   # Return something
10  return
11
12
13 def func_to_plot_the_data(*Arguments*):
14   # Do something with arguments
15   # Return something
16   return
```

```
[ ] 1 filepath =
2 filename =
3
4 # Read the data
5 data = func_to_read_the_data()
6
7 # Any additional reading statements
8
```

```
[ ] 1 # Analyze the data
2 analysis = func_to_analyze_the_data()
3
4 # Any additional analysis
5
```

```
[ ] 1 # Save/plot the data
2 func_to_plot_the_data
3
```

```
[ ] 1 # Any additional plotting
```

Set up your system and install any needed packages

Import your packages/modules (including any custom modules)

Mount your drive

Create different functions for frequently used code

It usually helps to create functions that do a specific tasks
(e.g. read, analyze, plot)

Read the data

Analyze the data

Plot the data

One cell per plot helps with organization

```
[ ] 1 # # Run this code only once per notebook, then comment it out
[ ] 2 # !pip install netcdf4
```

```
[ ] 1 import numpy as np
[ ] 2 import xarray as xr
[ ] 3 import matplotlib.pyplot as plt
[ ]
[ ] 5 import sys
[ ] 6 sys.path.append('/content/drive/My Drive')
[ ] 7 import my_module
[ ]
[ ] 9 from google.colab import drive
[ ] 10 drive.mount('/content/drive')
```

```
[ ] 1 def func_to_read_the_data(*Arguments*):
[ ] 2     # Do something with arguments
[ ] 3     # Return something
[ ] 4     return
[ ]
[ ] 6
[ ] 7 def func_to_analyze_the_data(*Arguments*):
[ ] 8     # Do something with arguments
[ ] 9     # Return something
[ ] 10    return
[ ]
[ ] 12
[ ] 13 def func_to_plot_the_data(*Arguments*):
[ ] 14     # Do something with arguments
[ ] 15     # Return something
[ ] 16     return
```

```
[ ] 1 filepath =
[ ] 2 filename =
[ ]
[ ] 4 # Read the data
[ ] 5 data = func_to_read_the_data()
[ ]
[ ] 7 # Any additional reading statements
[ ] 8
```

```
[ ] 1 # Analyze the data
[ ] 2 analysis = func_to_analyze_the_data()
[ ]
[ ] 4 # Any additional analysis
[ ] 5
```

```
[ ] 1 # Save/plot the data
[ ] 2 func_to_plot_the_data
[ ] 3
```

```
[ ] 1 # Any additional plotting
```

Consistent code style helps keep everything organized

PEP 8 -- Style Guide for Python Code

[https://www.python.org/
dev/peps/pep-0008/
#code-lay-out](https://www.python.org/dev/peps/pep-0008/#code-lay-out)

- [Introduction](#)
- [A Foolish Consistency is the Hobgoblin of Little Minds](#)
- [Code Lay-out](#)
 - [Indentation](#)
 - [Tabs or Spaces?](#)
 - [Maximum Line Length](#)
 - [Should a Line Break Before or After a Binary Operator?](#)
 - [Blank Lines](#)
 - [Source File Encoding](#)
 - [Imports](#)
 - [Module Level Dunder Names](#)
- [String Quotes](#)
- [Whitespace in Expressions and Statements](#)
 - [Pet Peeves](#)
 - [Other Recommendations](#)
- [When to Use Trailing Commas](#)
- [Comments](#)
 - [Block Comments](#)
 - [Inline Comments](#)
 - [Documentation Strings](#)
- [Naming Conventions](#)
 - [Overriding Principle](#)
 - [Descriptive: Naming Styles](#)
 - [Prescriptive: Naming Conventions](#)
 - [Names to Avoid](#)
 - [ASCII Compatibility](#)
 - [Package and Module Names](#)
 - [Class Names](#)
 - [Type Variable Names](#)
 - [Exception Names](#)
 - [Global Variable Names](#)
 - [Function and Variable Names](#)
 - [Function and Method Arguments](#)
 - [Method Names and Instance Variables](#)
 - [Constants](#)
 - [Designing for Inheritance](#)
 - [Public and Internal Interfaces](#)
- [Programming Recommendations](#)
 - [Function Annotations](#)
 - [Variable Annotations](#)
- [References](#)
- [Copyright](#)

Consistent code style helps keep everything organized

Blank Lines

Surround top-level function and class definitions with two blank lines.

Method definitions inside a class are surrounded by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Imports

- Imports should usually be on separate lines:

```
# Correct:  
import os  
import sys
```

```
# Wrong:  
import sys, os
```

It's okay to say this though:

```
# Correct:  
from subprocess import Popen, PIPE
```

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.

Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.

You should use two spaces after a sentence-ending period in multi-sentence comments, except after the final sentence.

Ensure that your comments are clear and easily understandable to other speakers of the language you are writing in.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Consistent code style helps keep everything organized

Whitespace in Expressions and Statements

- Immediately inside parentheses, brackets or braces:

```
# Correct:  
spam(ham[1], {eggs: 2})
```

```
# Wrong:  
spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parenthesis:

```
# Correct:  
foo = (0,)
```

```
# Wrong:  
bar = (0, )
```

- Immediately before a comma, semicolon, or colon:

```
# Correct:  
if x == 4: print x, y; x, y = y, x
```

```
# Wrong:  
if x == 4 : print x , y ; x , y = y , x
```

- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted:

```
# Correct:  
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:  
ham[lower:upper], ham[lower:upper:], ham[lower::step]  
ham[lower+offset : upper+offset]  
ham[: upper_fn(x) : step_fn(x)], ham[: step_fn(x)]  
ham[lower + offset : upper + offset]
```

```
# Wrong:  
ham[lower + offset:upper + offset]  
ham[1: 9], ham[1 :9], ham[1:9 :3]  
ham[lower : : upper]  
ham[ : upper]
```

- More than one space around an assignment (or other) operator to align it with another:

```
# Correct:  
x = 1  
y = 2  
long_variable = 3
```

```
# Wrong:  
x           = 1  
y           = 2  
long_variable = 3
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
# Correct:  
spam(1)
```

```
# Wrong:  
spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
# Correct:  
dct['key'] = lst[index]
```

```
# Wrong:  
dct ['key'] = lst [index]
```

More than one space around an assignment (or other) operator to align it with another:

```
# Correct:  
x = 1  
y = 2  
long_variable = 3
```

Consistent code style helps keep everything organized

```
1 import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Packages

A package is a directory of Python modules.

We have already seen a number of packages!

```
import matplotlib.pyplot as plt
```

To use the modules in a package you have to import it.

Useful packages for Oceanography

Already in Colab

1. numpy
2. scipy
3. xarray
4. Pandas
5. datetime
6. matplotlib.pyplot

Not in Colab

1. netCDF4
2. Cartopy
3. cmocean
4. Gibbs Seawater
(gsw)

How to find what packages Colab already has

!pip list

Package	Version	chainer	7.4.0	dcopt	0.6.2	gin-config	0.3.0	httpplib2	0.17.4	jpeg4py	0.1.4
<hr/>											
absl-py	0.10.0	charDET	3.0.4	docutils	0.16	glob2	0.7	httpplib2shim	0.0.3	jsonschema	2.6.0
alabaster	0.7.12	cloudpickle	1.3.0	dopamine-rl	1.0.5	google	2.0.3	humanize	0.5.1	jupyter	1.0.0
albumentations	0.1.12	cmake	3.12.0	earthengine-api	0.1.238	google-api-core	1.16.0	hyperopt	0.1.2	jupyter-client	5.3.5
altair	4.1.0	cmdstanpy	0.9.5	easydict	1.9	google-api-python-client	1.7.12	ideep4py	2.0.0.po	jupyter-console	5.2.0
argon2-cffi	20.1.0	colorlover	0.3.0	ecos	2.0.7.post	google-auth	1.17.2	idna	2.10	jupyter-core	4.7.0
asgiref	3.3.1	community	1.0.0b1	editdistance	0.5.3	google-auth-httplib2	0.0.4	image	1.5.33	jupyterlab-pygments	0.1.2
astor	0.8.1	contextlib2	0.5.5	en-core-web-sm	2.2.5	google-auth-oauthlib	0.4.2	imageio	2.4.1	kaggle	1.5.9
astropy	4.1	convertdate	2.3.0	entrypoints	0.3	google-cloud-bigquery	1.21.0	imagesize	1.2.0	kapre	0.1.3.1
astunparse	1.6.3	coverage	3.7.1	ephem	3.7.7.1	google-cloud-bigquery-storage	1.1.0	imbalanced-learn	0.4.3	Keras	2.4.3
async-generator	1.10	coveralls	0.5	et-xmlfile	1.0.1	google-cloud-core	1.0.3	imblearn	0.0	Keras-Preprocessing	1.1.2
atari-py	0.2.6	cromod	1.7	fa2	0.3.5	google-cloud-datastore	1.8.0	imgaug	0.2.9	keras-vis	0.4.1
atomicwrites	1.4.0	cufflinks	0.17.3	fancyimpute	0.4.3	google-cloud-firebase	1.7.0	importlib-metadata	2.0.0	kiwisolver	1.3.1
attrrs	20.3.0	cvxopt	1.2.5	fastdtw	0.3.4	google-cloud-language	1.2.0	importlib-resources	3.3.0	knimpute	0.1.0
audioread	2.1.9	cvxpy	1.0.31	fastprogress	1.0.0	google-cloud-storage	1.18.1	imutils	0.5.3	korean-lunar-calendar	0.2.1
autograd	1.3	cycler	0.10.0	fastlock	0.5	google-colab	1.0.0	iniflect	2.1.0	librosa	0.6.3
Babel	2.9.0	cymem	2.0.4	fbprophet	0.7.1	google-pasta	0.2.0	iniconfig	1.1.1	lightgbm	2.2.3
backcall	0.2.0	Cython	0.29.21	feather-format	0.4.1	google-resumable-media	0.4.1	intel-openmp	2020.0.1	llvmlite	0.31.0
beautifulsoup4	4.6.3	daft	0.0.4	filelock	3.0.12	googleapis-common-protos	1.52.0	ipykernel	4.10.1	lucid	0.3.8
bleach	3.2.1	dask	2.12.0	firebase-admin	4.4.0	googledrivedownloader	0.4	ipython	5.5.0	LunarCalendar	0.0.9
bilis	0.4.1	dataclasses	0.8	fix-yahoo-finance	0.0.22	graphviz	0.10.1	ipython-genutils	0.2.0	lxml	4.2.6
bokeh	2.1.1	datascience	0.10.6	Flask	1.1.2	grpcio	1.33.2	ipython-sql	0.3.9	Markdown	3.3.3
Bottleneck	1.3.2	debugpy	1.0.0	flatbuffers	1.12	gspread	3.0.1	ipywidgets	7.5.1	MarkupSafe	1.1.1
branca	0.4.1	decorator	4.4.2	folium	0.8.3	gspread-dataframe	3.0.8	itsdangerous	1.1.0	matplotlib	3.2.2
bs4	0.0.1	defusedxml	0.6.0	future	0.16.0	gym	0.17.3	jax	0.2.6	matplotlib-venn	0.11.6
CacheControl	0.12.6	descartes	1.1.0	gast	0.3.3	h5py	2.10.0	jaxlib	0.1.57+cu	missingno	0.4.2
cachetools	4.1.1	dill	0.3.3	GDAL	2.2.2	HeapDict	1.0.1	jdcal	1.4.1	mistune	0.8.4
catalogue	1.0.0	distributed	1.25.3	gdown	3.6.4	holidays	0.10.3	jedi	0.17.2	mizani	0.6.0
certifi	2020.11.8	Django	3.1.3	gensim	3.6.0	holoviews	1.13.5	jieba	0.42.1	mkl	2019.0
cffi	1.14.3	dlib	19.18.0	geographiclib	1.50	html5lib	1.0.1	Jinja2	2.11.2	mlxtend	0.14.0
cftime	1.3.0	dm-tree	0.1.5	geopy	1.17.0	httpimport	0.5.18	joblib	0.17.0	more-itertools	8.6.0
<hr/>											
moviepy	0.2.3.5	pandas	1.1.4	pyasn1	0.9.0	python-slugify	4.0.1	sortedcontainers	2.3.0	tifffile	2020.9.3
mpmath	1.1.0	pandas-datareader	0.9.0	py	0.13.3	python-utils	2.4.0	spacy	2.2.4	toml	0.10.2
msgpack	1.0.0	pandas-profiling	1.4.1	pyarrow	0.14.1	pytz	2018.9	Sphinx	1.8.5	toolz	0.11.1
multiprocess	0.70.11.1	pandocfilters	1.4.3	pyasn1	0.4.8	sphinxcontrib-serializinghtml	1.1.4	sphinxcontrib-websupport	1.2.4	torch	1.7.0+cu10
multitasking	0.0.9	panel	0.9.7	pyasn1-modules	0.2.8	PyWavelets	1.1.1	SQLAlchemy	1.3.20	torchtext	0.3.1
murmurhash	1.0.4	param	1.10.0	pycocotools	2.0.2	PyYAML	3.13	sqlparse	0.4.1	torchvision	0.8.1+cu10
music21	5.5.0	parso	0.7.1	pycparser	2.20	pymq	20.0.0	statsmodels	0.10.2	tqdm	4.41.1
natsort	5.5.0	pathlib	1.0.1	pyct	0.4.8	qtconsole	4.7.7	sympy	1.1.1	traitlets	4.3.3
nbclient	0.5.1	patsy	0.5.1	pydata-google-auth	1.1.0	QtPy	1.9.0	tables	3.4.4	tweepy	3.6.0
nbconvert	5.6.1	pexpect	4.8.0	pydot	1.3.0	regex	2019.12	tabulate	0.8.7	typeguard	2.7.1
nbformat	5.0.8	pickleshare	0.7.5	pydot-ng	2.0.0	requests	2.23.0	tbleib	1.7.0	typing-extensions	3.7.4.3
nest-asyncio	1.4.3	Pillow	7.0.0	pydotplus	2.0.2	requests-oauthlib	1.3.0	tensorboard	2.3.0	tzlocal	1.5.1
netCDF4	1.5.4	pip	19.3.1	PyDrive	1.3.1	resampy	0.2.2	tensorboard	1.5.1	umap-learn	0.4.6
networkx	2.5	pip-tools	4.5.1	pyemd	0.5.1	retrying	1.3.3	tensorboard-plugin-wit	1.7.0	uritemplate	3.0.1
nibabel	3.0.2	plac	1.1.3	pyglet	1.5.0	rpy2	3.2.7	tensorflow	0.0.22	urllib3	1.24.3
nltk	3.2.5	plotly	4.4.1	Pygments	2.6.1	rsa	4.6	tensorflow-addons	0.8.3	vega-datasets	0.8.0
notebook	5.3.1	plotnine	0.6.0	pygobject	3.26.1	scikit-image	0.16.2	tensorflow-datasets	4.0.1	wasabi	0.8.0
np-utils	0.5.12.1	pillow	7.0.0	pymc3	3.7	scikit-learn	0.22.2	tensorflow-estimator	2.3.0	wcwidth	0.2.5
numba	0.48.0	portpicker	1.3.1	pymeeus	0.3.7	scipy	1.4.1	tensorflow-gcs-config	2.3.0	webencodings	0.5.1
numexpr	2.7.1	prefetch-generator	1.0.1	pymongo	3.11.1	screen-resolution-extra	0.0.0	tensorflow-hub	0.10.0	Werkzeug	1.0.1
numpy	1.18.5	preshed	3.0.4	pymystem3	0.2.0	scs	2.1.2	tensorflow-metadata	0.25.0	wheel	0.35.1
nvidia-ml-py3	7.352.0	prettytable	2.0.0	PyOpenGL	3.1.5	seaborn	0.11.0	tensorflow-privacy	0.2.2	widgentsbextension	3.5.1
oauth2client	4.1.3	progressbar2	3.38.0	pyparsing	2.4.7	Send2Trash	1.5.0	tensorflow-probability	0.11.0	wordcloud	1.5.0
oauthlib	3.1.0	prometheus-client	0.9.0	pysistent	0.17.3	setuptools	50.3.2	termcolor	1.1.0	wrapt	1.12.1
okgrade	0.4.3	promise	2.3	pysndfile	1.3.8	setuptools-git	1.2	terminado	0.9.1	xarray	0.15.1
opencv-contrib-python	4.1.2.30	prompt-toolkit	1.0.18	PySocks	1.7.1	Shapely	1.7.1	testpath	0.4.4	xgboost	0.90
opencv-python	4.1.2.30	protobuf	3.12.4	pystan	2.19.1.1	simplegeneric	0.8.1	text-unicode	1.3	xkit	0.0.0
openpyxl	2.5.9	psutil	5.4.8	pytest	3.6.4	six	1.15.0	textblob	0.15.3	xlrd	1.1.0
opt-einsum	3.3.0	psycopg2	2.7.6.1	python-apt	1.6.5+ubun	sklearn	0.0	textgenrn	1.4.1	xlwt	1.3.0
osqp	0.6.1	ptyprocess	0.6.0	python-chess	0.23.11	sklearn-pandas	1.8.0	slugify	0.0.1	Theano	1.0.5
packaging	20.4	pv	1.9.0	python-dateutil	2.8.1	smart-open	3.0.0	smart-open	1.0.5	Thaanno	0.0.1
palettatable	3.3.0	python-clustering	3.3.0	python-louvain	0.14	smar	3.0.0	smar	3.0.0	Thaanno	0.0.1

How to import external packages to Colab

```
!pip install <package name>
```

To figure out the name to import the package, you can check the documentation! (Example: <https://matplotlib.org/cmocean/>)

GSW and cmocean

!pip install gsw

!pip install cmocean

```
1 # Package management
2 # Run this cell only once per notebook then comment it out.
3 #-----
4
5 # This code allows xarray and netCDF4 to work with Google Colab
6 !pip install netcdf4
7
8 # This code installs TEOS-10 gsw
9 !pip install gsw
10
11 # This code installs cmocean
12 !pip install cmocean
13
14 # This code allows cartopy to work with Google Colab
15 !grep '^deb ' /etc/apt/sources.list | \
16   sed 's/^deb /deb-src /g' | \
17   tee /etc/apt/sources.list.d/deb-src.list
18 !apt-get -qq update
19 !apt-get -qq build-dep python3-cartopy
20 !pip uninstall -y shapely
21 !pip install shapely --no-binary shapely
22 !pip install Cartopy
```

```
1 import numpy as np
2 import xarray as xr
3 import pandas as pd
4 import gsw
5 from scipy import stats, interpolate
6 from datetime import datetime, timedelta
7
8 import matplotlib.pyplot as plt
9 import cmocean
10
11 import cartopy.crs as ccrs
12 import cartopy.feature as cfeature
13 import matplotlib.ticker as mticker
14 from cartopy.mpl.gridliner import LONGITUDE_FORMATTER, LATITUDE_FORMATTER
```

GSW and cmocean

!pip install gsw

!pip install cmocean

```
1 # # Package management
2 # # Run this cell only once per notebook then comment it out.
3 # -----
4
5 # # This code allows xarray and netCDF4 to work with Google Colab
6 # !pip install netcdf4
7
8 # # This code installs TEOS-10 gsw
9 # !pip install gsw
10
11 # # This code installs cmocean
12 # !pip install cmocean
13
14 # # This code allows cartopy to work with Google Colab
15 # !grep '^deb ' /etc/apt/sources.list | \
16 #   sed 's/^deb /deb-src /g' | \
17 #   tee /etc/apt/sources.list.d/deb-src.list
18 # !apt-get -qq update
19 # !apt-get -qq build-dep python3-cartopy
20 # !pip uninstall -y shapely
21 # !pip install shapely --no-binary shapely
22 # !pip install Cartopy
```

```
1 import numpy as np
2 import xarray as xr
3 import pandas as pd
4 import gsw
5 from scipy import stats, interpolate
6 from datetime import datetime, timedelta
7
8 import matplotlib.pyplot as plt
9 import cmocean
10
11 import cartopy.crs as ccrs
12 import cartopy.feature as cfeature
13 import matplotlib.ticker as mticker
14 from cartopy.mpl.gridliner import LONGITUDE_FORMATTER, LATITUDE_FORMATTER
```

How to find what modules are in a package

dir(<package name>)

```
1 dir(gsw)
```

```
['CT_first_derivatives',
 'CT_first_derivatives_wrt_t_exact',
 'CT_freezing',
 'CT_freezing_first_derivatives',
 'CT_freezing_first_derivatives_poly',
 'CT_freezing_poly',
 'CT_from_enthalpy',
 'CT_from_enthalpy_exact',
 'CT_from_entropy',
 'CT_from_pt',
 'CT_from_rho',
 'CT_from_t',
 'CT_maxdensity',
 'CT_second_derivatives',
 'C_from_SP',
 'Fdelta',
 'Helmholtz_energy_ice',
 'Hill_ratio_at_SP2',
 'IPV_vs_fNsquared_ratio',
 'Nsquared',
 'O2sol',
 'O2sol_SP_pt',
 'SAAR',
 'SA_freezing_from_CT',
 'SA_freezing_from_CT_poly',
 'SA_freezing_from_t',
 'SA_freezing_from_t_poly',
 'SA_from_SP',
 'SA_from_SP_Baltic',
 'SA_from_Sstar',
 'SA_from_rho',
 'SP_from_C',
 'SP_from_SA',
 'SP_from_SA_Baltic',
 'SP_from_SK',
 'SP_from_SR',
 'SP_from_Sstar',
 'SP_salinometer',
 'SR_from_SP',
 'Sstar_from_SA',
 'Sstar_from_SP',
```

How to find what modules are in a package

**More information on each functions can
be found at the documentation:**

https://teos-10.github.io/GSW-Python/gsw_flat.html

**Or you can get information on a specific
function using ?**

gsw.sigma0?

Help X

Signature: gsw.sigma0([SA](#), [CT](#))
Docstring:
Calculates potential density anomaly with reference pressure
of 0 dbar,
this being this particular potential density minus 1000
kg/m^3. This
function has inputs of Absolute Salinity and Conservative
Temperature.
This function uses the computationally-efficient expression
for
specific volume in terms of SA, CT and p (Roquet et al.,
2015).
Parameters

SA : array-like
 Absolute Salinity, g/kg
CT : array-like
 Conservative Temperature (ITS-90), degrees C
Returns

sigma0 : array-like, kg/m^3
 potential density anomaly with
 respect to a reference pressure of 0 dbar,
 that is, this potential density - 1000 kg/m^3.
File: /usr/local/lib/python3.6/dist-
packages/gsw/_wrapped_ufuncs.py
Type: function

What we'll cover in this lesson

1. Creating functions
2. Organizing your code and installing external packages
- 3. String formatting**

Formatting strings to fit your needs

Formatting strings makes hard-coding unnecessary

```
1 T = 10  
2  
3 print('The temperature is 10°C.')
```

The temperature is 10°C.

```
1 T = 10  
2  
3 print('The temperature is '+str(T)+'°C.')
```

The temperature is 10°C.

<string>.format()

```
1 T = 10  
2  
3 print('The temperature is {}'.format(T))
```

The temperature is 10°C

Formatting strings to fit your needs

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(T,RH))
```

Temperature = 10°C, Relative humidity= 52%

Formatting strings to fit your needs

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(T,RH))
```

Temperature = 10°C, Relative humidity= 52%

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(RH,T))
```

Temperature = 52°C, Relative humidity= 10%

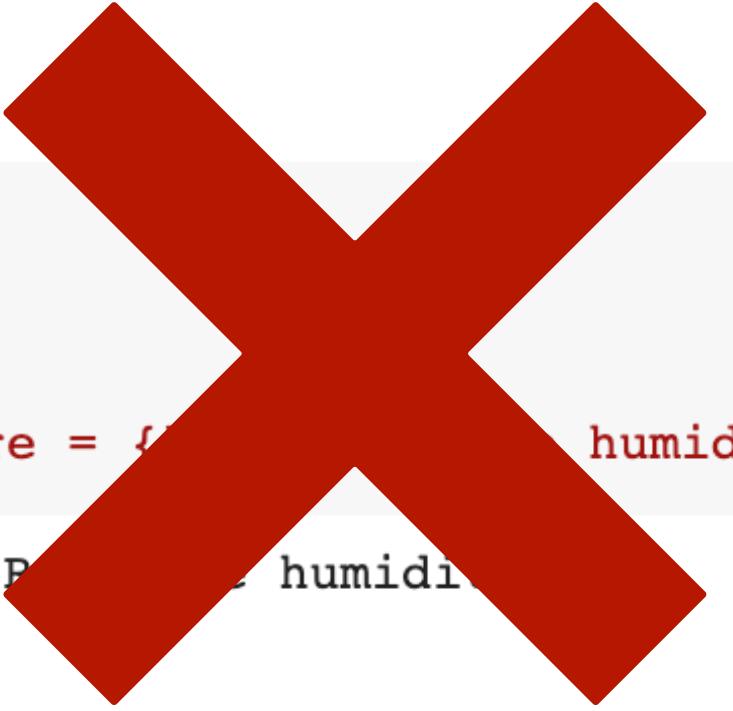
Formatting strings to fit your needs

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(T,RH))
```

Temperature = 10°C, Relative humidity= 52%

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(RH,T))
```

Temperature = 52°C, Relative humidity= 10%



```
1 T = 10
2 RH = 52
3
4 print('Temperature = {1}°C, Relative humidity= {0}%'.format(RH,T))
```

Temperature = 10°C, Relative humidity= 52%

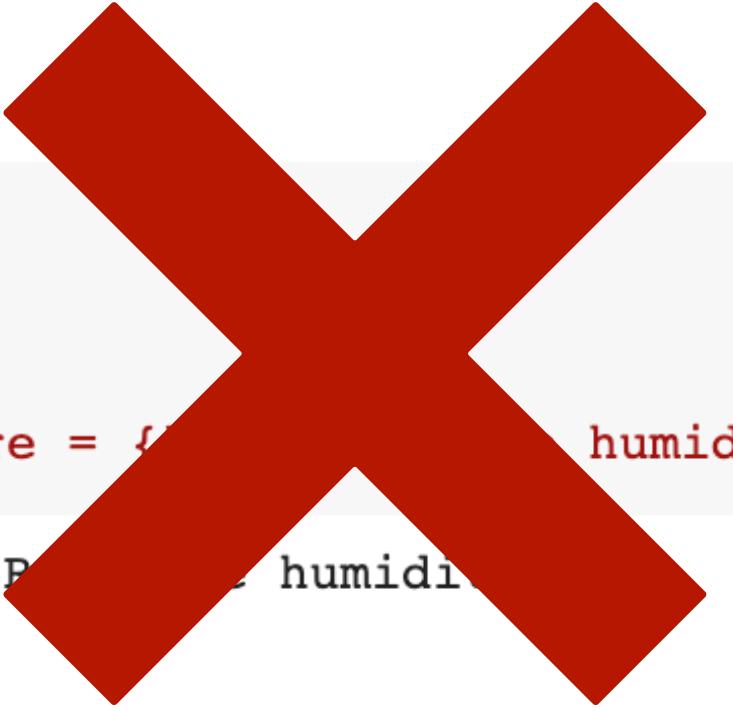
Formatting strings to fit your needs

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(T,RH))
```

Temperature = 10°C, Relative humidity= 52%

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {}°C, Relative humidity= {}%'.format(RH,T))
```

Temperature = 52°C, Relative humidity= 10%



```
1 T = 10
2 RH = 52
3
4 print('Temperature = {1}°C, Relative humidity= {0}%'.format(RH,T))
```

Temperature = 10°C, Relative humidity= 52%

```
1 T = 10
2 RH = 52
3
4 print('Temperature = {T}°C, Relative humidity= {RH}%'.format(RH=RH,T=T))
```

Temperature = 10°C, Relative humidity= 52%

You can round the output of a float

```
1 T = 10.71717
2 RH = 52.517985
3
4 print('Temperature = {T}°C, Relative humidity= {RH}%'.format(RH=RH,T=T))
```

```
Temperature = 10.71717°C, Relative humidity= 52.517985%
```

```
1 T = 10.71717
2 RH = 52.517985
3
4 print('Temperature = {T:.2f}°C, Relative humidity= {RH:.4f}%'.format(RH=RH,T=T))
```

```
Temperature = 10.72°C, Relative humidity= 52.5180%
```

You can customize the outputs

This is the string This is the function These are the inputs
`'{0:6.3f}'.format(3.141592653589)`

You can customize the outputs

The diagram illustrates the components of a string formatted with the `.format()` method. A horizontal line represents the string: `'{0:6.3f}'.format(3.141592653589)`. Three annotations with arrows point to specific parts:

- An arrow points to the placeholder `{0:6.3f}` with the label "This is the placeholder".
- An arrow points to the function call `.format(` with the label "This is the function".
- An arrow points to the input value `3.141592653589` with the label "These are the inputs".

This is the
placeholder

- **blank:** the format arguments are in order
- **integer:** the format argument position is known
- **name:** the format argument has keyword arguments

You can customize the outputs

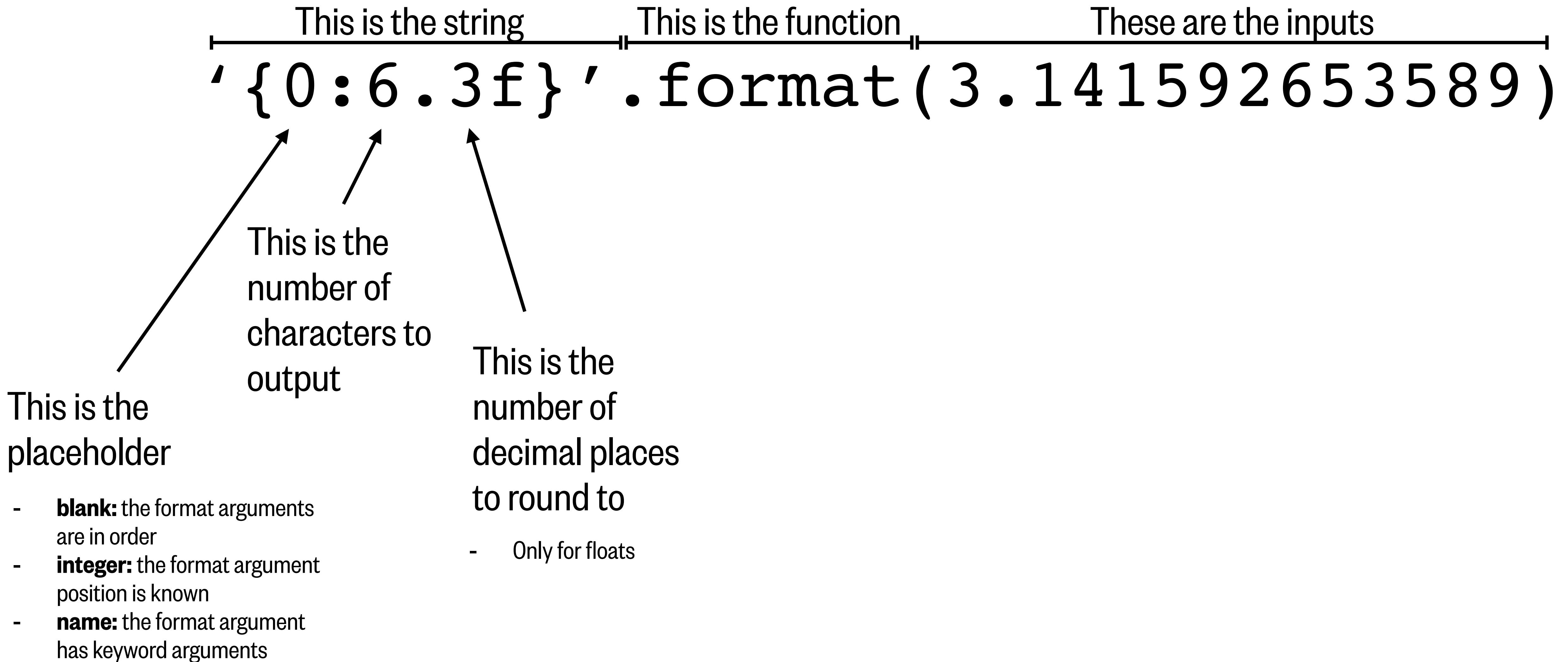
'{0:6.3f}'.format(3.141592653589)

This is the
number of
characters to
output

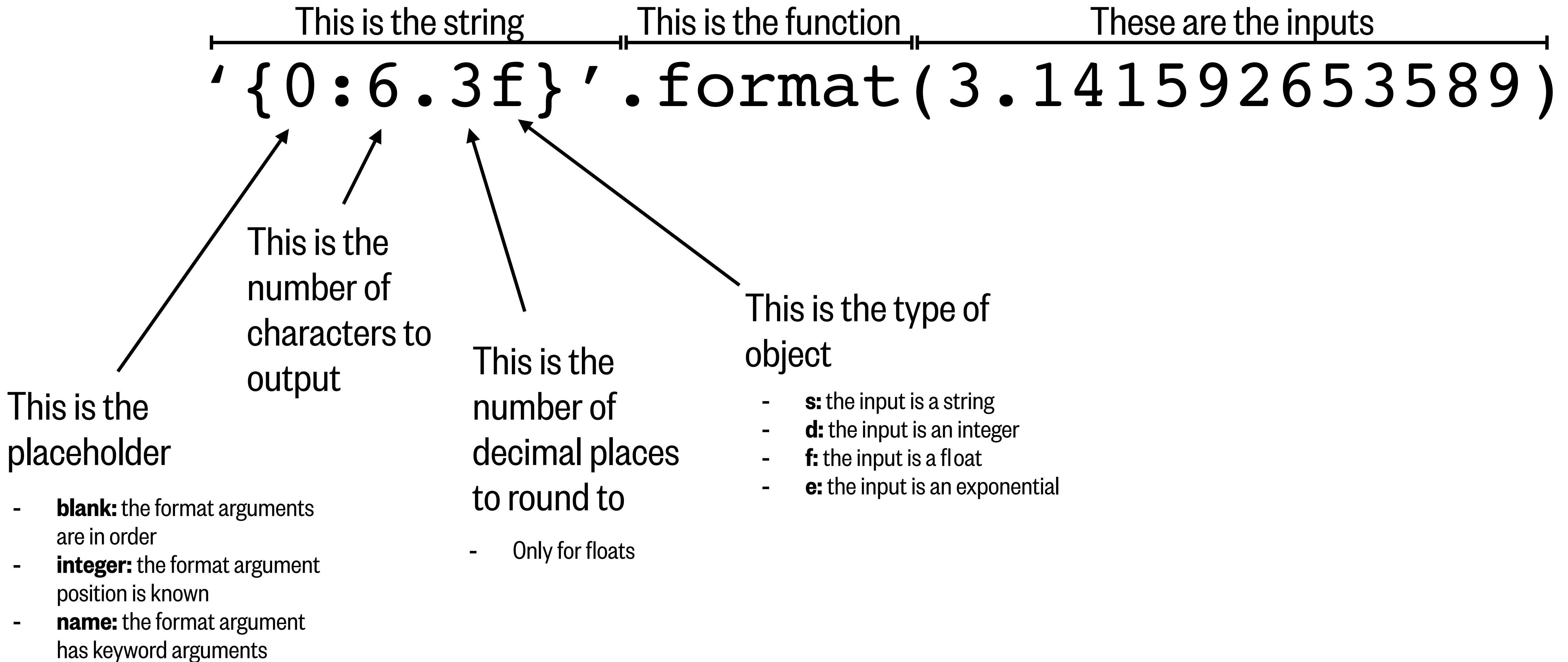
This is the
placeholder

- **blank:** the format arguments are in order
- **integer:** the format argument position is known
- **name:** the format argument has keyword arguments

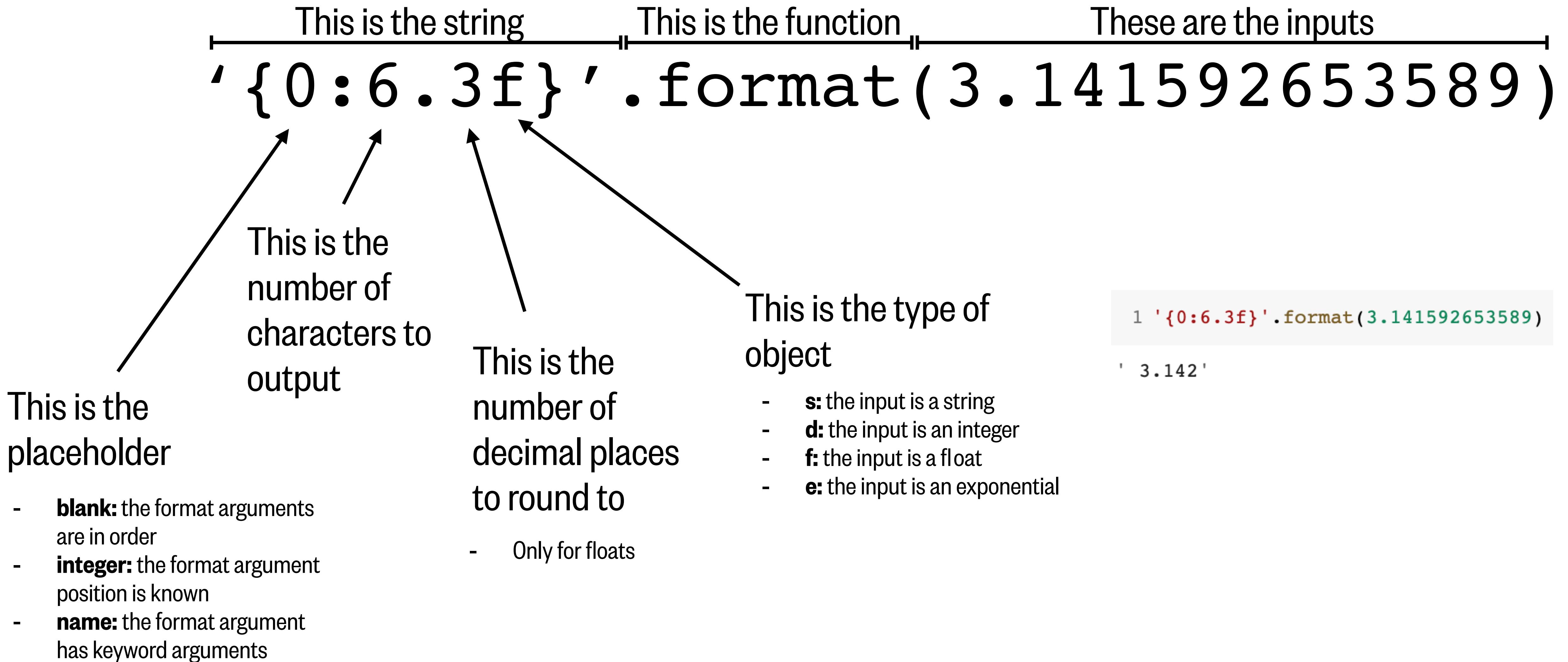
You can customize the outputs



You can customize the outputs



You can customize the outputs



You can customize the outputs

This is the string This is the function These are the inputs
`'{0:6.3f}'.format(3.141592653589)`

Additional
formatting can
go here.

**These are only
relevant to
number inputs**

>	Right-aligned
<	Left-aligned
^	Center-aligned
0	Zero-padded
space	A space is in front of positive numbers, a negative sign is in front of negative numbers
+	A positive sign is in front of positive numbers, a negative sign is in front of negative numbers

String formatting is particularly helpful in for loops

```
1 filenames = [ 'D5901105_185.nc', 'D5901105_186.nc', 'D5901105_187.nc',
2                 'D5901105_188.nc', 'D5901105_189.nc', 'D5901105_190.nc' ]
3
```

```
1 filenames = []
2 for index in range(185,191):
3     floatstring = 'D5901105_{:03d}.nc'.format(index)
4     filenames.append(floatstring)
5
6 print(filenames)
```

```
['D5901105_185.nc', 'D5901105_186.nc', 'D5901105_187.nc', 'D5901105_188.nc', 'D5901105_189.nc', 'D5901105_190.nc']
```

String formatting is particularly helpful in for loops

```
1 filenames = [ 'D5901105_185.nc', 'D5901105_186.nc', 'D5901105_187.nc',
2                 'D5901105_188.nc', 'D5901105_189.nc', 'D5901105_190.nc' ]
3
```

```
1 filenames = []
2 for index in range(185,191):
3     floatstring = 'D5901105_{:03d}.nc'.format(index)
4     filenames.append(floatstring)
5
6 print(filenames)
```

```
['D5901105_185.nc', 'D5901105_186.nc', 'D5901105_187.nc', 'D5901105_188.nc', 'D5901105_189.nc', 'D5901105_190.nc']
```

```
1 filenames = []
2 for index in range(1,7):
3     floatstring = 'D5901105_{:03d}.nc'.format(index)
4     filenames.append(floatstring)
5
6 print(filenames)
```

```
['D5901105_001.nc', 'D5901105_002.nc', 'D5901105_003.nc', 'D5901105_004.nc', 'D5901105_005.nc', 'D5901105_006.nc']
```

String formatting can also be helpful in functions

```
1 def read_profile(floatnum, profnum, filepath):
2     profstring = 'D{0:d}_{1:03d}.nc'.format(floatnum, profnum)
3     print(profstring)
4     data = xr.open_dataset(filepath+profstring)
5     data = data.squeeze()
6     return data
7
8 filepath = 'drive/My Drive/Data_folder/ARGO_floatdata/'
9 data = read_profile(5901105, 186, filepath)
10 display(data)
```

D5901105_186.nc

xarray.Dataset

► Dimensions: (N_HISTORY: 4, N_LEVELS: 76, N_PARAM: 4)

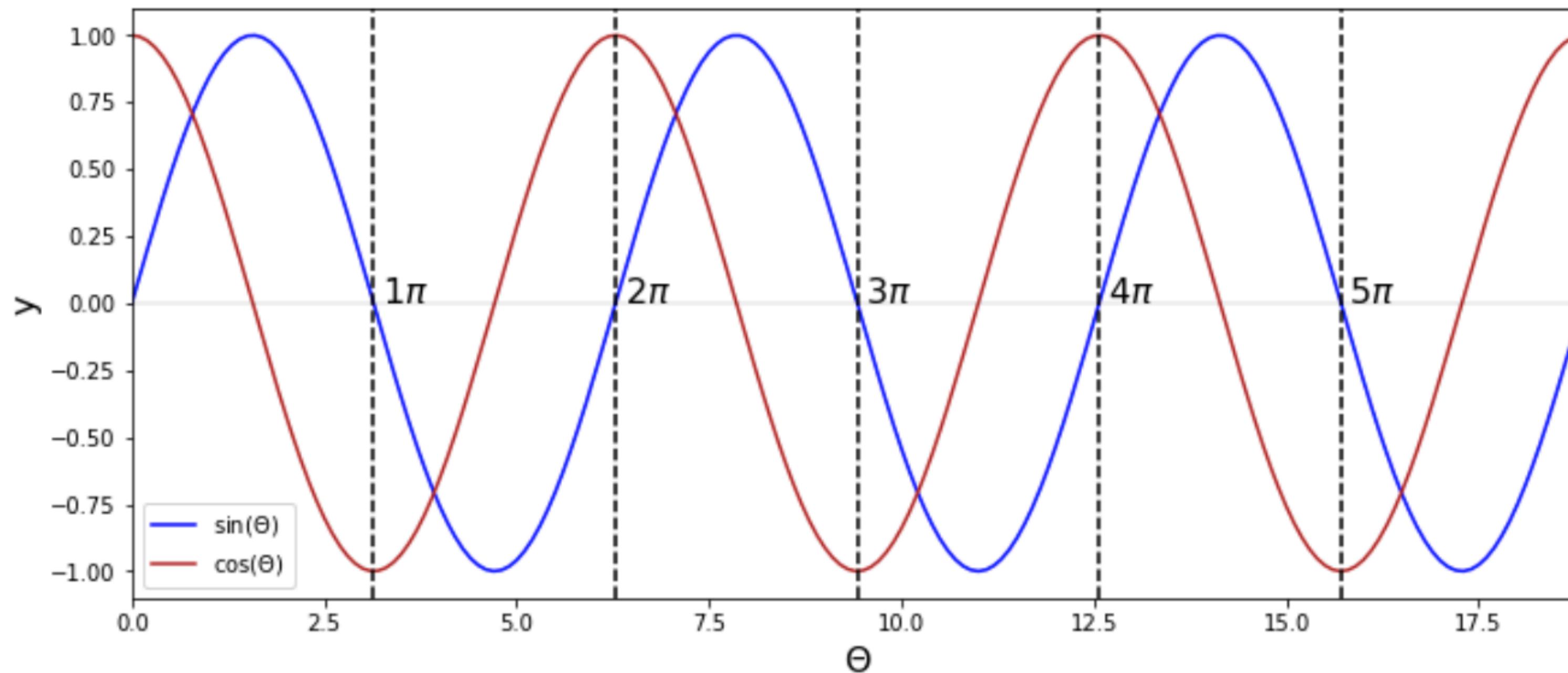
► Coordinates: (0)

► Data variables: (70)

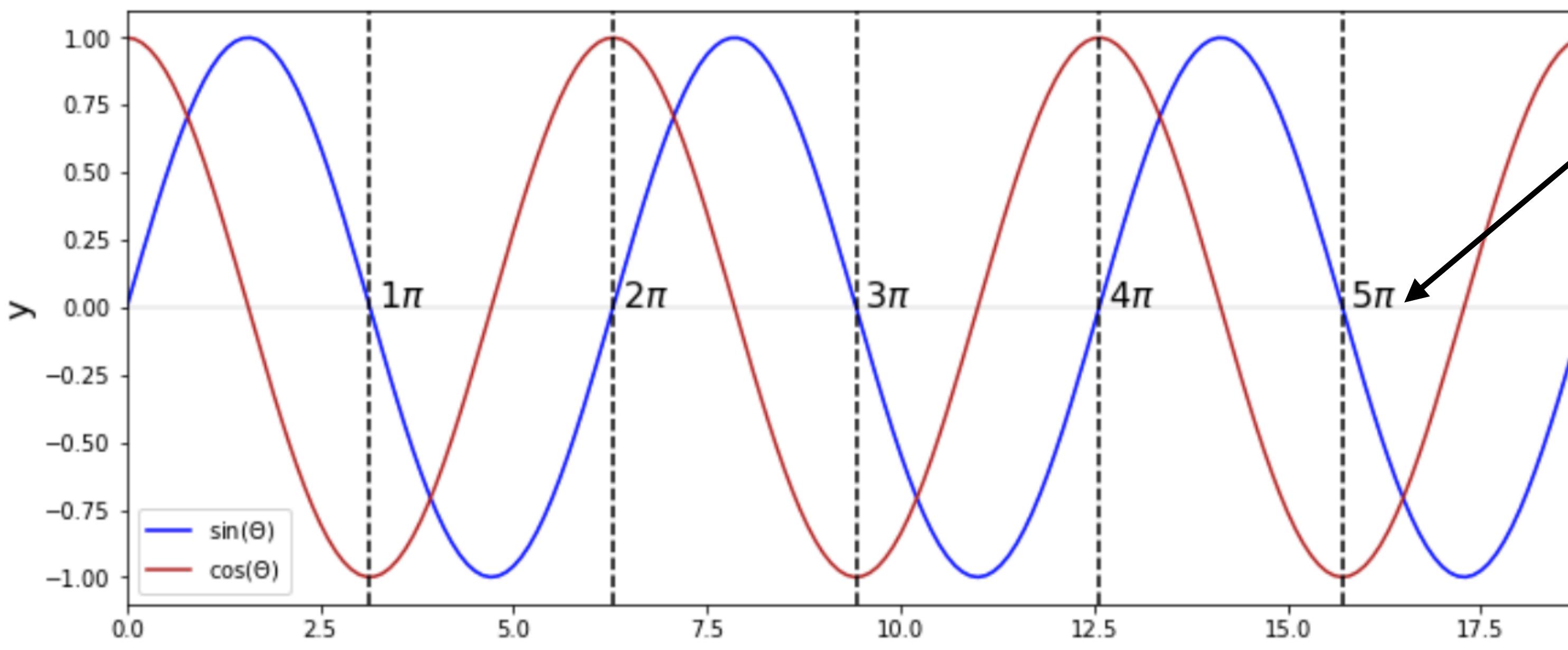
▼ Attributes:

title :	Argo float vertical profile
institution :	AOML
source :	Argo float
history :	2019-02-26T17:55:57Z creation
references :	http://www.argodatamgt.org/Documentation
comment :	free text
user_manual_v...	3.2
Conventions :	Argo-3.2 CF-1.6
featureType :	trajectoryProfile

Adding math symbols to strings on figures



Adding math symbols to strings on figures



' \$\\Theta\$ '

' \$\\pi\$ '

```
1 theta = np.arange(0,6*np.pi,0.1)
2 y1 = np.sin(theta)
3 y2 = np.cos(theta)
4
5 fs = 16
6
7 fig = plt.figure(figsize=(12,5))
8 plt.plot(theta,y1,'b',label='sin($\\Theta$)')
9 plt.plot(theta,y2,c='firebrick',label='cos($\\Theta$)')
10
11 plt.plot([0,6*np.pi],[0,0],'k-',lw=0.25,alpha=0.5)
12
13 for index in range(1,6):
14     plt.plot([index*np.pi, index*np.pi], [-1.1,1.1], 'k--')
15     plt.text(index*np.pi+0.1,0, '{}$\\pi$'.format(index), fontsize=fs)
16
17 plt.xlim([0,6*np.pi])
18 plt.ylim([-1.1,1.1])
19 plt.xlabel('$\\Theta$', fontsize=fs)
20 plt.ylabel('y', fontsize=fs)
21
22 plt.legend()
23
```

Adding math symbols to figures

Super- and sub- scripts

x^2	O_2	π	σ	Δ	κ	α
β	ρ	ω	∞	\sum	\pm	

\$ \$
↑
Insert character text here

For more special symbols, see the documentation

<https://matplotlib.org/3.1.1/tutorials/text/mathtext.html>