Python - Builtins, Modules, Packages

```
module = Module(
    code="ELEE1147",
    name="Programming for Engineers",
    credits=15,
    module_leader="Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA"
)
```

UNIVERSITY OF GREENWICH

Download as a PDF

Builtins

```
$ python
Python 3.12.8 (tags/v3.12.8:2dc476b, Dec  3 2024, 19:30:04) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dir(__builtins__)
```

```
ArithmeticError
AssertionError
AttributeError
BaseException
BaseExceptionGroup
BlockingIOError
BrokenPipeError
BufferError
BytesWarning
ChildProcessError
ConnectionAbortedError
ConnectionError
ConnectionRefusedError
ConnectionResetError
DeprecationWarning
EOFError
Ellipsis
EncodingWarning
EnvironmentError
Exception
ExceptionGroup
False
FileExistsError
FileNotFoundError
FloatingPointError
FutureWarning
GeneratorExit
IOError
ImportError
ImportWarning
IndentationError
IndexError
InterruptedError
IsADirectoryError
KeyError
KeyboardInterrupt
LookupError
MemoryError
ModuleNotFoundError
NameError
None
NotADirectoryError
NotImplemented
NotImplementedError
OSError
OverflowError
PendingDeprecationWarning
PermissionError
ProcessLookupError
RecursionError
ReferenceError
ResourceWarning
RuntimeError
RuntimeWarning
StopAsyncIteration
StopIteration
SyntaxError
SyntaxWarning
SystemError
SystemExit
TabError
TimeoutError
True
TypeError
UnboundLocalError
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
UnicodeWarning
UserWarning
ValueError
Warning
WindowsError
ZeroDivisionError
__build_class__
__debug__
__doc__
__import__
__loader__
__name__
__package__
__spec__
abs
aiter
all
anext
any
ascii
bin
bool
breakpoint
bytearray
bytes
callable
chr
classmethod
compile
complex
copyright
credits
delattr
dict
dir
divmod
enumerate
eval
exec
exit
filter
float
format
frozenset
getattr
globals
hasattr
hash
help
hex
id
input
int
isinstance
issubclass
iter
len
license
list
locals
map
max
memoryview
min
next
object
oct
open
ord
pow
print
property
quit
range
repr
reversed
round
set
setattr
slice
sorted
staticmethod
str
sum
super
tuple
type
vars
zip
```

```python
def main():
    # abs() - absolute value
    print("abs(-42):", abs(-42))

    # all() - check if all elements are True
    print("all([True, True, False]):", all([True, True, False]))

    # any() - check if any element is True
    print("any([0, 0, 1]):", any([0, 0, 1]))

    # bin() - convert integer to binary string
    print("bin(10):", bin(10))

    # bool() - get boolean value
    print("bool('Hello'):", bool("Hello"))

    # chr() - get character from ASCII code
    print("chr(65):", chr(65))

    # divmod() - quotient and remainder
    print("divmod(17, 5):", divmod(17, 5))

    # enumerate() - get index and value
    for idx, value in enumerate(['apple', 'banana', 'cherry']):
        print(f"Item {idx}: {value}")

    # eval() - evaluate a string as Python expression
    expression = "3 * (4 + 5)"
    print(f"eval('{expression}'):", eval(expression))

    # filter() - filter items
    numbers = [1, 2, 3, 4, 5]
    even = list(filter(lambda x: x % 2 == 0, numbers))
    print("Even numbers:", even)

if __name__ == "__main__":
    main()
```

Modules

```
$ python
Python 3.12.8 (tags/v3.12.8:2dc476b, Dec  3 2024, 19:30:04) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help('modules')
```

Python - Builtins, Modules, Packages > Builtins > Modules

```
3c22db4583604893 51e4__mypyc
PIL
__future__
__hello__
__phello__
_aix_support
_asyncio
_bz2
_collections_abc
_compat_pickle
_compression
_ctypes
_ctypes_test
_decimal
_distutils_hack
_elementtree
_hashlib
_lzma
_markupbase
_msi
_multiprocessing
_osx_support
_overlapped
_py_abc
_pydatetime
_pydecimal
_pyio
_pylong
_pytest
_queue
_sitebuiltins
_socket
_sqlite3
_ssl
_strptime
_testbuffer
_testcapi
_testclinic
_testconsole
_testimportmultiple
_testinternalcapi
_testmultiphase
_testsinglephase
_threading_local
_tkinter
_uuid
_version
_weakrefset
_wmi
_yaml
_zoneinfo
abc
aifc
annotated_types
antigravity
argparse
ast
astroid
asyncio
base64
bdb
bibtexparser
bisect
build
builtins
bz2
cProfile
cachecontrol
calendar
certifi
cgi
cgitb
charset_normalizer
chunk
cleo
cmd
code
codecs
codeop
collections
colorama
colorsys
compileall
concurrent
configparser
contextlib
contextvars
contourpy
copy
copyreg
coverage
crashtest
crypt
csv
ctypes
curses
cycler
dataclasses
datetime
dateutil
dbm
deadcode
decimal
difflib
dill
dis
distlib
doctest
docutils
dulwich
email
encodings
ensurepip
enum
et_xmlfile
fastjsonschema
feedparser
filecmp
fileinput
filelock
fitz
fnmatch
fontTools
fractions
ftplib
functools
genericpath
getopt
getpass
gettext
glob
graphlib
gzip
hashlib
heapq
hmac
html
http
id
idlelib
idna
imaplib
imghdr
importlib
iniconfig
init_py_project
inspect
installer
io
ipaddress
isort
json
keyring
keyword
kiwisolver
lib2to3
linecache
linkify_it
locale
logging
lxml
lzma
mailbox
mailcap
markdown_it
matplotlib
mccabe
mdit_py_plugins
mdurl
mimetypes
modulefinder
modules
more_itertools
msgpack
msilib
multiprocessing
netrc
nh3
nntplib
ntpath
nturl2path
numbers
numpy
opcode
openpyxl
operator
optparse
os
packaging
pandas
pathlib
pdb
pickle
pickletools
pip
pipes
pkg_resources
pkginfo
pkgutil
platform
platformdirs
plistlib
pluggy
poplib
posixpath
pprint
profile
pstats
psutil
pty
py
py_compile
pyclbr
pydantic
pydantic_core
pydoc
pydoc_data
pyexpat
pygments
pylab
pylings
pylint
pymupdf
pyparsing
pyproject_hooks
pyproject_toml
pytest
pytz
pyzotero
queue
quopri
random
rapidfuzz
re
readme_renderer
remarks
reprlib
requests
requests_toolbelt
rfc3986
rich
rlcompleter
runpy
sched
seaborn
secrets
select
selectors
setuptools
sgmllib
shapely
shellingham
shelve
shlex
shutil
signal
site
six
smtplib
sndhdr
socket
socketserver
sqlite3
sre_compile
sre_constants
sre_parse
ssl
stat
statistics
string
stringprep
struct
subprocess
sunau
symtable
sysconfig
tabnanny
tarfile
telnetlib
tempfile
test
textual
textwrap
this
threading
timeit
tkinter
token
tokenize
toml
tomli
tomlkit
tomllib
tqdm
trace
traceback
tracemalloc
trove_classifiers
tty
turtle
turtledemo
twine
types
typing
typing_extensions
tzdata
uc_micro
unicodedata
unittest
urllib
urllib3
uu
uuid
venv
virtualenv
warnings
watchdog
wave
weakref
webbrowser
webdav3
wheel
win32ctypes
winsound
workbook
wsgiref
xdrlib
xlrd
xlutils
xlwt
xml
xmlrpc
yaml
zipapp
zipfile
zipimport
zoneinfo
```

337 Modules

Modules in Python

- Code reusability:
  - Write once, use many times
  - contains collections of functions and global variables
  - Modules can be imported into other Python files
- Maintainability:
  - Easier to manage and update code
- Namespace management:
  - Avoids name conflicts by separating code logically

- To use a module, you my first `import` it

```python
import math
import random
import datetime
import os
import sys

def main():
    # math module: calculate square root
    num = 25
    print(f"Square root of {num} is {math.sqrt(num)}")

    # random module: generate random integer
    rand_num = random.randint(1, 100)
    print(f"Random number between 1 and 100: {rand_num}")

    # datetime module: get current date and time
    now = datetime.datetime.now()
    print(f"Current date and time: {now}")

    # os module: get current working directory
    cwd = os.getcwd()
    print(f"Current working directory: {cwd}")

    # sys module: show Python version
    print(f"Python version: {sys.version}")

    # os + random: create a random filename
    filename = f"file_{random.randint(1000,9999)}.txt"
    full_path = os.path.join(cwd, filename)
    print(f"Generated random file path: {full_path}")

if __name__ == "__main__":
    main()
```

- To use a module's member(s)/function(s)/variable(s), you use the `.` operator

```python
import math
import random
import datetime
import os
import sys

def main():
    # math module: calculate square root
    num = 25
    print(f"Square root of {num} is {math.sqrt(num)}")

    # random module: generate random integer
    rand_num = random.randint(1, 100)
    print(f"Random number between 1 and 100: {rand_num}")

    # datetime module: get current date and time
    now = datetime.datetime.now()
    print(f"Current date and time: {now}")

    # os module: get current working directory
    cwd = os.getcwd()
    print(f"Current working directory: {cwd}")

    # sys module: show Python version
    print(f"Python version: {sys.version}")

    # os + random: create a random filename
    filename = f"file_{random.randint(1000,9999)}.txt"
    full_path = os.path.join(cwd, filename)
    print(f"Generated random file path: {full_path}")

if __name__ == "__main__":
    main()
```

- To use a module's member(s)/function(s)/variable(s), you use the `.` operator
- You can also access at a greater depth too

```python
import math
import random
import datetime
import os
import sys

def main():
    # math module: calculate square root
    num = 25
    print(f"Square root of {num} is {math.sqrt(num)}")

    # random module: generate random integer
    rand_num = random.randint(1, 100)
    print(f"Random number between 1 and 100: {rand_num}")

    # datetime module: get current date and time
    now = datetime.datetime.now()
    print(f"Current date and time: {now}")

    # os module: get current working directory
    cwd = os.getcwd()
    print(f"Current working directory: {cwd}")

    # sys module: show Python version
    print(f"Python version: {sys.version}")

    # os + random: create a random filename
    filename = f"file_{random.randint(1000,9999)}.txt"
    full_path = os.path.join(cwd, filename)
    print(f"Generated random file path: {full_path}")

if __name__ == "__main__":
    main()
```

- The reusability, means we can call the same function over and over again!

```python
import math
import random
import datetime
import os
import sys

def main():
    # math module: calculate square root
    num = 25
    print(f"Square root of {num} is {math.sqrt(num)}")

    # random module: generate random integer
    rand_num = random.randint(1, 100)
    print(f"Random number between 1 and 100: {rand_num}")

    # datetime module: get current date and time
    now = datetime.datetime.now()
    print(f"Current date and time: {now}")

    # os module: get current working directory
    cwd = os.getcwd()
    print(f"Current working directory: {cwd}")

    # sys module: show Python version
    print(f"Python version: {sys.version}")

    # os + random: create a random filename
    filename = f"file_{random.randint(1000,9999)}.txt"
    full_path = os.path.join(cwd, filename)
    print(f"Generated random file path: {full_path}")

if __name__ == "__main__":
    main()
```

User-defined Modules

1. Write functions and variables in a `.py` file, e.g., `math_utils.py`, `stats_utils.py`
2. Import them in another script using e.g., `import math_utils`

```
# Directory structure
*
|-main.py
|-math_utils.py
|-stats_utils.py
```

```python
# math_utils.py
def mean(numbers):
    """Calculate the arithmetic mean."""
    return sum(numbers) / len(numbers)
```

```python
# stats_utils.py
def mean(numbers):
    """Calculate the geometric mean."""
    product = 1
    for number in numbers:
        product *= number
    return product ** (1 / len(numbers))
```

```python
# main.py
import math_utils
import stats_utils

numbers = [1, 2, 3, 4, 5]

# Using the arithmetic mean from math_utils
arithmetic_mean = math_utils.mean(numbers)
print("Arithmetic Mean:", arithmetic_mean)

# Using the geometric mean from stats_utils
geometric_mean = stats_utils.mean(numbers)
print("Geometric Mean:", geometric_mean)
```

Example of Pylings

- You should always try have a top level `docstrings`, inline with PEP.

```
"""
logging_setup.py: Configures logging for the Pylings application.

This module defines the `setup_logging` function, which enables file-based
logging when the application is run in debug mode. Logs are written to a
predefined file path (`DEBUG_PATH`) and include timestamps, severity levels,
source modules, and message content.

Usage:
    Call `setup_logging(debug=True)` early in the application to enable
    debug-level logging to file.

Intended for use by CLI entry points and debugging support.
"""
import logging
from pylings.constants import DEBUG_PATH

def setup_logging(debug: bool):
    """Configure application-wide logging based on debug flag.

    If debug mode is enabled, logs are written to a file defined by `DEBUG_PATH`.
    The log format includes timestamps, log level, module, and message.

    Args:
        debug (bool): Whether to enable detailed logging to file.
    """
    handlers = []

    if debug:
        handlers.append(logging.FileHandler(DEBUG_PATH, mode="w"))

        logging.basicConfig(
            level=logging.DEBUG,
            format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
            handlers=handlers
        )
# End-of-file (EOF)
```

Example of Pylings

- Import Builtin modules first

```
"""
logging_setup.py: Configures logging for the Pylings application.

This module defines the `setup_logging` function, which enables file-based
logging when the application is run in debug mode. Logs are written to a
predefined file path (`DEBUG_PATH`) and include timestamps, severity levels,
source modules, and message content.

Usage:
    Call `setup_logging(debug=True)` early in the application to enable
    debug-level logging to file.

Intended for use by CLI entry points and debugging support.
"""
import logging
from pylings.constants import DEBUG_PATH

def setup_logging(debug: bool):
    """Configure application-wide logging based on debug flag.

    If debug mode is enabled, logs are written to a file defined by `DEBUG_PATH`.
    The log format includes timestamps, log level, module, and message.

    Args:
        debug (bool): Whether to enable detailed logging to file.
    """
    handlers = []

    if debug:
        handlers.append(logging.FileHandler(DEBUG_PATH, mode="w"))

        logging.basicConfig(
            level=logging.DEBUG,
            format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
            handlers=handlers
        )
# End-of-file (EOF)
```

Example of Pylings

- Import user-defined modules second

```
"""
logging_setup.py: Configures logging for the Pylings application.

This module defines the `setup_logging` function, which enables file-based
logging when the application is run in debug mode. Logs are written to a
predefined file path (`DEBUG_PATH`) and include timestamps, severity levels,
source modules, and message content.

Usage:
    Call `setup_logging(debug=True)` early in the application to enable
    debug-level logging to file.

Intended for use by CLI entry points and debugging support.
"""
import logging
from pylings.constants import DEBUG_PATH

def setup_logging(debug: bool):
    """Configure application-wide logging based on debug flag.

    If debug mode is enabled, logs are written to a file defined by `DEBUG_PATH`.
    The log format includes timestamps, log level, module, and message.

    Args:
        debug (bool): Whether to enable detailed logging to file.
    """
    handlers = []

    if debug:
        handlers.append(logging.FileHandler(DEBUG_PATH, mode="w"))

        logging.basicConfig(
            level=logging.DEBUG,
            format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
            handlers=handlers
        )
# End-of-file (EOF)
```

Example of Pylings

- `docstrings` used for functions too.

```
"""
logging_setup.py: Configures logging for the Pylings application.

This module defines the `setup_logging` function, which enables file-based
logging when the application is run in debug mode. Logs are written to a
predefined file path (`DEBUG_PATH`) and include timestamps, severity levels,
source modules, and message content.

Usage:
    Call `setup_logging(debug=True)` early in the application to enable
    debug-level logging to file.

Intended for use by CLI entry points and debugging support.
"""
import logging
from pylings.constants import DEBUG_PATH

def setup_logging(debug: bool):
    """Configure application-wide logging based on debug flag.

    If debug mode is enabled, logs are written to a file defined by `DEBUG_PATH`.
    The log format includes timestamps, log level, module, and message.

    Args:
        debug (bool): Whether to enable detailed logging to file.
    """
    handlers = []

    if debug:
        handlers.append(logging.FileHandler(DEBUG_PATH, mode="w"))

        logging.basicConfig(
            level=logging.DEBUG,
            format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
            handlers=handlers
        )
# End-of-file (EOF)
```

Third-Party Modules

Installing

- `pip` - Pip Installs Packages [`p`ackage `i`nstaller for `p`ython]
  - looks locally first
  - then looks at https://pypi.org/{package}
- ~630k packages!

```
curl https://pypi.org/simple/ | sed -E 's/<[^>]+>//g' > packages.txt && wc -l packages.txt
```

- Use `pip`, the package installer for Python

```
pip --help
pip install <package>
pip install <package>==<version>
```

Checking Installed Modules

- Use the following command to see installed packages

```
$ pip list

    Package            Version
------------------ -----------
alabaster          1.0.0
asgiref            3.8.1
babel              2.16.0
build              1.2.2.post1
certifi            2024.12.14
charset-normalizer 3.4.1
colorama           0.4.6
Django             5.1.6
docutils           0.21.2
id                 1.5.0
...                ...
```

- Or check specific package information

```
$ pip show requests

Name: requests
Version: 2.31.0
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: path/to/file
Requires: idna, charset-normalizer, certifi, urllib3
Required-by:
```

Installing packages from a file

- `-r`: Install from the given requirements file. This option can be used multiple times.

```
pip install -r requirements.txt
```

- `requirements.txt`: could be any name but the file holds your packages to be installed

```
# requirements.txt
requests==2.25.1
numpy>=1.21.0
pandas
flask==2.0.1
```

Renaming a Module

- The module name is long or cumbersome to type repeatedly.
- There’s a naming conflict with another module or variable.
- You want a shorter or more intuitive name for readability.
- Use `as` to give a module a different name in your code
- You can also import specific functions or variables

```
from math import pi, sqrt # builtins
import matplotlib.pyplot as plt # third-party
import mymodule # user defined package

# Now you can use 'plt' instead of 'matplotlib.pyplot'
print(pi)             # Output: 3.14159...
print(sqrt(16))       # Output: 4.0
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```