Compressed Python:

A summary-rundown for techs with no time

- Tristan Mendoza - Jan 06, 2025

Python Data Types Overview

Type	Bracket	Example	Update	Access	Quick definition
Lists	square	[2, 3, "a"]	mutable	sequence	Ordered list, mixed types, duplicates allowed
Tuple	parens	(2, 3, "a")	immutable	sequence	Ordered: hold constants, is hashable
Sets	curlies	{'b', 3, 'h'}	both	direct	Mixed bag (unordered)
Dicts	curlies	{"a":1, "b":2}	mutable	mapping	Key-value pairs, (unordered)

Type	Declaration	Type check	Display example	Casting notes
List	list_a = [1, 2, 3]	type(list_a) → list	list_a → [1, 2, 3]	From any iterable (tuple or set)
Tuple	tuple_a = (4, 5, 6)	type(tuple_a) → tuple	tuple_a → (4, 5, 6)	From any iterable (list or set)
Set	set_a = {7, 8, 9}	type(set_a) → set	set_a → {7, 8, 9}	Any iterable (list or set), zaps duplicates
Dict	dict_a = {0: "a", 1: "b"}	type(dict_a) → dict	dict_a → {0: 'a', 1: 'b'}	Needs an iterable of key-value pairs

Creating a new empty set (all other types are straightforward):

```
>>> tester1=().   # makes empty tuple
>>> tester2={}.   # makes empty dict
```

>>> tester3= set()
>>> type(tester3)
<class 'set'>

Sequence operations +, * [,] work with tuples, lists and strings

The + operator creates a new tuple as the concatenation of the arguments

```
>>>  ("chapter",8) + ("strings","tuples","lists")
('chapter', 8, 'strings', 'tuples', 'lists')
```

The * operator between a tuple and a number creates a new tuple with repetitions of the input tuple.

```
>>> 2*(3,"blind","mice")
(3, 'blind', 'mice', 3, 'blind', 'mice')
```

The simple [ ] operator selects an item or a slice

Get an item out

Use list1[index] to refer to a single item. with items are numbered from 0 onward.
A negative index starts counting from the end backwards (so -1 is the last item)

```
>>> listA=[10,20,30,40,50,60]
>>> listA[0]          - returns 10
>>> listA[-1]         - returns 60
```

Splitting and slicing (for lists, tuples, strings)

To split a sequence in two at any index x: my_list[:x] and my_list[x:]

```
>>> listA[:2]          # split up to 2 - returns [10, 20]
>>> listA[2:]          # split starting from 2 - returns [30, 40, 50, 60]
```

Strides: slicing based on intervals (for lists, tuples, strings)

The form s[a:b:c] can be used to specify a stride or step c, causing the resulting slice to skip items.
The stride can also be negative, returning items in reverse.

```
>>> s = 'bicycle'
>>> s[::3]          - returns 'bye'
>>> s[::-1]         - returns 'elcycib'
>>> s[::-2]         - returns 'eccb'

>>> a_string = 'My alphabet starts where your alphabet ends.'
>>> a_string[3:11]         ----- 3rd to 11th chars
'alphabet'
>>> a_string[3:-3]         ----- 3rd char to 3rd before end
'alphabet starts where your alphabet en'
>>> a_string[0:2]          ----- Begins at 0, up to but not including 2 (a space)
'My'
>>> a_string[:18]          ----- Implied beginning of string [:n] is the first n characters
'My alphabet starts'
>>> a_string[18:]          ----- Implied end of string [n:] is n till the end
' where your alphabet ends.'
```

Python expression operators

Operator	*Description*
x + y	Addition: Adds two values.
x - y	Subtraction: Subtracts one value from another.
x * y	Multiplication: Multiplies two values.
x / y	Division: Divides one value by another (float division).
x // y	Floor Division: Divides and truncates to the nearest integer.
x % y	Modulo: Returns the remainder of the division.
x ** y	Exponentiation: Raises x to the power of y.
x | y	Union: Combines elements (e.g., sets: `{1, 2} | {2, 3} → {1, 2, 3}`).
x & y	Intersection: Common elements (e.g., sets: `{1, 2} & {2, 3} → {2}`).
x - y	Difference: Elements in x but not in y (e.g., `{1, 2, 3} - {2} → {1, 3}`).
x ^ y	Symmetric Difference: Elements in either x or y but not both (e.g., `{1, 2} ^ {2, 3} → {1, 3}`).
x < y	Less than: True if x is less than y.
x > y	Greater than: True if x is greater than y.
x <= y	Less than or equal to: True if x is less than or equal to y.
x >= y	Greater than or equal to: True if x is greater than or equal to y.
x == y	Equality: True if x is equal to y.
x != y	Inequality: True if x is not equal to y.
x and y	Logical AND: True if both x and y are True.
x or y	Logical OR: True if either x or y is True.
not x	Logical NOT: True if x is False.
x if cond else y	Ternary conditional: Returns x if condition is True, otherwise returns y.
x << y	Left Shift: Shifts bits of x left by y places.
x >> y	Right Shift: Shifts bits of x right by y places.
x & y	Bitwise AND: Performs AND operation on bits.
x | y	Bitwise OR: Performs OR operation on bits.
x ^ y	Bitwise XOR: Performs XOR operation on bits.
~x	Bitwise NOT: Inverts the bits of x.
x in y	Membership: True if x is in y.
x not in y	Non-membership: True if x is not in y.
x is y	Identity: True if x and y reference the same object.
x is not y	Non-identity: True if x and y do not reference the same object.

The Walrus Operator

In Python 3.8+ the walrus operator := was introduced to assign values to variables as part of an expression (which is then called an assignment expression). Combining variable assignment and usage in one place is more efficient and readable.

```
### Example:
if (n := len(my_list)) > 5:
    print(f"The list is too long: {n} elements.")

###  Reduced Redundancy: Avoid recalculating values unnecessarily.
# Without walrus operator
n = len(my_list)
if n > 5:
    print(f"The list is too long: {n} elements.")

#  With walrus operator
if (n := len(my_list)) > 5:
    print(f"The list is too long: {n} elements.")

### Flexible Contexts: Use in loops, comprehensions, and function arguments.
# Loop example
while (line := input("Enter text: ")) != "quit":
    print(f"You entered: {line}")

#  Comprehension examples
filtered = [y for x in data if (y := process(x)) > 0]
#
positive_numbers = [num for num in data if (squared := num**2) > 100]
```

Can't be used outside of an expression since it's not expressly where it was designed to be used.
Great for where a value is needed both for conditional checks and further ops, keeping code readable, but succinct

Virtual environments for projects - Setting up your workspace

A virtual environment is a lightweight Python installation with its own package directories and a Python binary copied (or linked) create it, with advantages of ensuring integrity and making sure installed packages don't collide.

Recommended: create a main directory where you keep all the project environments.
Placing this directory in your project tree has efficiency issues.

virtualenv is classic - can be used with Python 2 and 3
pyvenv was designed for Python3- it's a wrapper around venv module and is now deprecated (in 3.6).
On Python >= 3.3 use venv module directly (virtualenv is still there)

Use the following to create a virtual environment called my_env1 depending on version:

```
$ virtualenv envs/my_env1                    # in Python 2
$ python3 -m venv envs/my_env1   # Python 3 to create, then activate it to work in it
$ source envs/my_env/bin/activate  # Prompt changes to show you're in your env
(your_env) $
```

The last part of running the activate script is more convenient than changing the $PATH.
Now pip and python commands are localized
Using pip without having to sudo ensures you don't escalate privileges of things it installs
Your modules/packages from various projects won't contaminate each other or conflict.

If you haven't activated the environment you can still work within the environment running commands and scripts

```
$ /home/user5/envs/my_env/bin/python myscript.py
$ /home/user5/envs/my_env/bin/pip
```

For venv and virtualenv, the only other really useful command is deactivate. Verbose for anything is -v. There is one special option that needs mentioning. It might be a pain to install a big list of packages that are already on the host system, and you can get it use the host system's stuff with "python3 -m venv --system-site-packages envs/my_env", just don't do it. It will mess things up in production, your environment won't contain those packages- it will put you in a world of hurt. *Just don't.* I'm reading a advanced Python book that mentions this option without pointing out how stupid it is.

You need to specify the Python version (interpreter) for a project?

```
$ python3.8 -m venv envs/your_env
$ virtualenv -p python3.8 envs/your_env
```

You could do this but having multiple installs of Python on your host OS can be problematic and gets "old fast".
Installing pyenv is a remedy for this [https://github.com/pyenv/pyenv-installer]
Enable it's virtualenv functionality with pyenv- virtualenv [https://github.com/pyenv/pyenv-virtualenv]
As seen below, it grabs the versions you tell it and puts them in your home directory to use when necessary

```
$ pyenv install --list                    # Nice big list to choose from
$ pyenv install --list | grep 3.3         # Where's the one I need?
$ pyenv install 3.3                       # Make it so!
---- > Installed Python-3.3 to /home/user5/.pyenv/versions/3.3
```

So now, with the virtualenv functionality built you just do this:

```
$ pyenv virtualenv 3.3 /envs/my_env
$ pyenv activate my_pyenv
```

Commands for pyenv-virtualenv

Install or uninstall a specific Python version	pyenv install (uninstall) version
List all available Python versions	pyenv install --list
Show current Python version	pyenv version
List all installed Python versions	pyenv versions
Rebuild the shim cache	pyenv rehash
Check for and apply updates to pyenv	pyenv update
Check for missing dependencies	pyenv doctor
Install with a patch	pyenv install --patch version
Create a virtual environment	pyenv virtualenv version env_name
List all virtual environments	pyenv virtualenvs
Activate a virtual environment	pyenv activate env_name
Deactivate the current virtual environment	pyenv deactivate
Delete a virtual environment	pyenv uninstall env_name
Find the location of a Python executable	pyenv which command_name
Run commands with a specific Python version	pyenv exec command
Show environment variables for a version	pyenv exec

Using PIP for package management

With both install and upgrade, <pkg_name>==<version> can be specified

Install package(s) Package name can be a URL or local file	pip install <pkg_name1>, <pkg_name2>
Install a specific version	pip install <pkg_name>==<version>
Install packages listed in file (use with pip freeze)	pip install -r requirements.txt
Upgrade a package	pip install --upgrade <pkg_name> (OR use -r requirements.txt)
Install package without its dependencies	pip install --no-deps <pkg_name>
Search for packages	pip search <search_term>
Download the package without installing it	pip download <pkg_name>
Get info about a package	pip show <pkg_name>
Generate requirements.txt listing installed packages	pip freeze > requirements.txt
List all installed packages (add -o for outdated pkgs)	pip list
Check for security issues, dependency problems	pip check OR pip verify
Show the dependency tree of installed packages	pipdeptree
Upgrade the pip installation itself	pip install --upgrade pip
Configure proxy and cache settings for pip	pip config
Create a wheel package for distribution	pip wheel <package_directory>

Using pipenv for managing virtual environments

A way of managing simpler projects is by using pipenv. For larger projects with development teams, poetry is better suited. By going into your project's directory and running 'pipenv install' pipenv will look at the location specified where store your environment files, match it to the directory of the project you are working in using hash mapping (of the directory)

- The default location fo where pipenv puts it's env directory is ~/.local/share/virtualenvs
- To set a preferred location for the virtual environment files, set the WORKON_HOME environment variable
  export WORKON_HOME=/path/to/my/custom/virtualenvs
- Use can use pipenv --venv to verify which virtual environment is associated with the current project directory
- After running 'pipenv install package-name>' in your project's directory, pipenv will create a pipfile and pifile.lock there.

Create virtual environment, install packages from Pipfile	pipenv install
Activate the virtual environment	pipenv shell
Install a package in the virtual environment	pipenv install <pkg_name>
Uninstall a package from the virtual environment	pipenv uninstall <pkg_name>
Generate a Pipfile.lock file with package versions	pipenv lock
Get more options to use	pipenv --help

Anaconda/conda packages

Anaconda/Conda is focused primarily on data science and machine learning packages and managing virtual environments, yet is now considered a bit sluggish and heavy on system resources. Mamba/MicroMamba is a faster solution for accessing the same repositories. Both have functionality for managing virtual environments, yet either pipenv or Poetry are much better for doing such things. Since we need something to access these repos for the packages that are offered (pip can't) , MicroMamaba is currently the best option. Most of the commands for conda and Mamba are the same as illustrated below.

Task	**Conda Command**	**Mamba Command**
Install a specific package	conda install <pkg_name>	mamba install <pkg_name>
Install multiple packages	conda install pkg1 pkg2	mamba install pkg1 pkg2
Install a specific version	conda install <pkg_name>=version	mamba install <pkg_name>=version
Install from a specific channel	conda install -c channel_name pkg	mamba install -c channel_name pkg
Install packages from a file	conda install --file file.txt	mamba install --file file.txt
Install without dependency checks	conda install <pkg_name> --no-deps	mamba install <pkg_name> --no-deps
Install into a specific prefix	conda install --prefix /path pkg	mamba install --prefix /path pkg
Dry-run installation	conda install <pkg_name> --dry-run	mamba install <pkg_name> --dry-run
Update a specific package	conda update <pkg_name>	mamba update <pkg_name>
Update all packages	conda update --all	mamba update --all
Remove a package	conda remove <pkg_name>	mamba remove <pkg_name>
Force remove a package	conda remove --force <pkg_name>	mamba remove --force <pkg_name>
List installed packages	conda list	mamba list
Export list of installed packages	conda list --export > file.txt	mamba list --export > file.txt
Check outdated packages	conda list --outdated	mamba list --outdated
Search for a package	conda search <pkg_name>	mamba search <pkg_name>
Check package details	conda search <pkg_name> --info	mamba search <pkg_name> --info
Query repository for dependencies	conda search <pkg_name>	mamba repoquery depends <pkg_name>
Find reverse dependencies	Not available	mamba repoquery whoneeds <pkg_name>
Clean unused cache/files	conda clean --all	mamba clean --all

Poetry for creating, updating, and sharing projects (including package management)

- Running "poetry init" asks a few questions to form a pyproject.toml - autofills the author name and email from the current user's git config (you can of course change it), package name, version, description, and prefered Python version to associate with. Your asked if you want to interactively define dependences and confirm to finish up.
- The common modern replacement for setup.py are *.toml files (Tom's Obvious Minimal Language)
- To enter the activated virtual environment when using poetry, you can simply type 'poetry shell'.
- When making an entry for a cron job inside an environment you would have to first have the command line change directory to the project, then issue a 'poetry run' command from that location.

When it finishes you add dependencies with 'poetry add <needed_package>', (name only is fine) and it will report it's installing (meaning it is adding it to the TOML file) and creates a lock file, which has more specific package info, the exact version the hash it should match, and the dependencies it needs. The idea of all of this is you can take these files to another system and run 'poetry install' and it will rebuild it for you in a virtual environment automatically. It also serves as a point of reference for upgrades. When you use 'poetry update' it will automatically check dependencies with versions of other items in the TOML to ensure compatibility.

- The default location poetry places and looks for virtual environment files is ~/.cache/pypoetry/virtualenvs
- Change this with 'export POETRY_VIRTUALENVS_PATH=/path/to/my/custom/virtualenvs"
- Verify this when needed with 'poetry env info --path'

Poetry Commands

Project Management Commands	
Create a new Poetry project	poetry new project_name
Initialize a project in the current directory	poetry init
Display Poetry version	poetry --version
Check current project dependencies	poetry check
Build a project package	poetry build
Virtual Environment Management Commands	
Create a virtual environment	Automatically handled by poetry install
List Poetry-managed virtual environments	poetry env list
Use a specific Python version	poetry env use python_version
Remove a virtual environment	poetry env remove python_version
Activate the virtual environment	poetry shell
Deactivate the virtual environment	exit (from within the shell)
Check virtual environment path	poetry env info --path
Package Management Commands	
Install dependencies	poetry install
Add a new package	poetry add <pkg_name>
Add a package with specific version	poetry add <pkg_name>@version
Add a dev dependency	poetry add <pkg_name> --group dev
Remove a package	poetry remove <pkg_name>
Update all packages	poetry update
Update a specific package	poetry update <pkg_name>
Check for outdated dependencies	poetry show --outdated
Show installed packages	poetry show
Search for a package	poetry search <pkg_name>
Lock dependencies	poetry lock
Export dependencies to requirements.txt	poetry export -f requirements.txt --output requirements.txt
Scripts and Execution Commands	
Run a command in the virtual environment	poetry run command
Run a defined script	poetry run script_name
List defined scripts	poetry run --list
Execute shell in the virtual environment	poetry shell
Publishing Commands	
Publish a package to a repository	poetry publish
Publish to a specific repository	poetry publish --repository repo_name
Build and publish in one command	poetry publish --build
Configuration Commands	
Set a configuration value	poetry config key value
View all configuration values	poetry config --list
Unset a configuration value	poetry config --unset key
Use local configuration	poetry config --local key value
Use global configuration	poetry config --global key value

List comprehensions

List comprehensions offer a concise syntax for generating lists. They allow for loops and optional conditions in a single line
List comprehensions load the entire list into memory. For large datasets, generator expressions may be more efficient.

```
###  Basic syntax
result = [x+1 for x in some_sequence]
###  Equivalent to:
result = [ ]
for x in some_sequence:
    result.append(x + 1)

###  With conditional:
result = [x + 1 for x in some_sequence if x > 23]
###  Equivalent to:
result = [ ]
for x in some_sequence:
    if x > 23:
        result.append(x + 1)

###  Nested loops (flattening a list of lists):
result = [x for sublist in list_of_lists for x in sublist]
###  Equivalent to:
result = [ ]
for sublist in list_of_lists:
    for x in sublist:
        result.append(x)

###  Example use cases:
[i for i in range(6)]                       # [0, 1, 2, 3, 4, 5] - simple range
[i * 2 for i in range(6)]                   # [0, 2, 4, 6, 8, 10] - transform values
[i for i in range(6) if i % 2 == 0]         # [0, 2, 4] - only even numbers
[i * 3 for i in range(6) if i % 2 == 0]     # [0, 6, 12, 18, 24] - even numbers transformed
```

Set comprehensions

Set comprehensions for creating sets- similar to list comprehensions but use braces `{}` instead of brackets `[]`

```
###  Basic Syntax
        Expression defines values; iterable is source of values to process; optional condition filters values included in the set
s = {expression for item in iterable if condition}

### Example: Unique values from a transformation
        Divides each number in range 0-9 by 2, keeping only unique results as a set; sorted() used to display set as a sorted list
s = {x // 2 for x in range(10)}
print(sorted(s))                            # Output: [0, 1, 2, 3, 4]

### Example: Squares of numbers
        Computes the square of each number in the range 0-9 and stores the unique results in a set.
squares = {x**2 for x in range(10)}
print(squares)                              # Output: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

### Example: Filtering even numbers
        Filters the range 0-9 to include only even numbers in the set.
evens = {x for x in range(10) if x % 2 == 0}
print(evens)                                # Output: {0, 2, 4, 6, 8}

### Example: Powers of two
        Generates powers of 2 from 2^0 to 2^9 and stores them in a set.
powers_of_two = {2**x for x in range(10)}
print(powers_of_two)                        # Output: {1, 2, 4, 8, 16, 32, 64, 128, 256, 512}
```

Dictionary Comprehensions

Similar to list comprehensions but with key-value pairs derived from iterables, a concise way to make dictionaries.

```
###  Basic Syntax:
d = {key: value for key, value in some_iterable}

###  Basic key-value mapping:
squares = {n: n * n for n in range(5)}
print(squares)                  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
###  File metadata dictionary:
import os, glob
metadata_dict = {f: os.stat(f) for f in glob.glob('*test*.py')}
print(metadata_dict)          # Output: { 'test1.py': os.stat_result(...), 'test2.py': os.stat_result(...) }

###  Filtered comprehension:
even_squares = {n: n * n for n in range(10) if n % 2 == 0}
print(even_squares)         # Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

###   Using a conditional to determine both keys and values:
parity = {n: 'even' if n % 2 == 0 else 'odd' for n in range(5)}
print(parity)                     # Output: {0: 'even', 1: 'odd', 2: 'even', 3: 'odd', 4: 'even'}

###   Handling nested dictionaries or lists:
nested_dict = {x: {y: x * y for y in range(3)} for x in range(3)}
print(nested_dict)            # Output: {0: {0: 0, 1: 0, 2: 0}, 1: {0: 0, 1: 1, 2: 2}, 2: {0: 0, 1: 2, 2: 4}}
```

Generators

Generators are a special type of iterable that allow lazy evaluation. Using the yield keyword, they produce items one at a time, pausing execution between each item while retaining their state. This makes them memory-efficient, especially for handling large or infinite datasets. They do not store the entire sequence in memory and rely on the next() function to resume execution until the next yield.

```
###  Example with yield
def count_up_to(n):
    for i in range(1, n + 1):
        yield i

counter = count_up_to(5)
print(list(counter))       # Output: [1, 2, 3, 4, 5]
```

Passing Values to Generators

Starting with Python 3.x, yield can accept values via send and return values when the generator terminates

```
def echo():
    while True:
        value = (yield)    # Receives value from g.send()
        print(f"Received: {value}")

g = echo()
next(g)                    # Prime the generator to reach the first yield
g.send("hello")          # Sends "hello" to the generator - Output is: Received: hello
```

yield from

The yield from statement (introduced in Python 3.3) delegates operations to the subiterable, automatically handling iteration and passing results back to the caller.

```
def combined_generator():
    yield from range(1, 4)   # Delegates to range(1, 4)
    yield from "abc"         # Delegates to string "abc"
    yield from [10, 20, 30]  # Delegates to a list

for item in combined_generator():
    print(item, end=" ")
# Output: 1 2 3 a b c 10 20 30
```

Example: Iterating using generators in loops

Generators integrate seamlessly with for loops, which automatically handle StopIteration when the generator is exhausted.

```
def reverse_string(my_str):
    for i in range(len(my_str) - 1, -1, -1):
        yield my_str[i]

###  Reverse a string using a generator
for char in reverse_string("Python"):
    print(char, end=" ")  # Output: n o h t y P
```

Example: Representing huge data streams
Generators handle infinite or large datasets efficiently by producing items on demand, minimizing memory usage.

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2

even_numbers = all_even()
print(next(even_numbers))  # Output: 0
print(next(even_numbers))  # Output: 2
```

Example: Pipelining generators
Generators can be used together in a pipeline to perform sequential operations. For example, one generator can produce numbers while another processes them, enabling efficient data processing.

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x + y
        yield x

def square(nums):
    for num in nums:
        yield num**2
# Sum of squares of Fibonacci numbers
print(sum(square(fibonacci_numbers(10))))  # Output: 4895
```

Generator expressions
Generator expressions are similar to list comprehensions but use parentheses () instead of brackets [ ]. They produce an iterator, yielding items one by one, making them more memory-efficient for large datasets.

Basic syntax
```
gen_exp = (x * x for x in range(10))
```

Example: Generator expressions as function arguments
When a generator expression is passed directly as a function argument, parentheses can be omitted for simplicity
```
my_list = [1, 3, 6, 10]
print(sum(x**2 for x in my_list))  # Output: 146
print(max(x**2 for x in my_list))  # Output: 100
```

Example: Calculate the sum of squares of single-digit integers
```
sum(x * x for x in range(10))
#  Output: 285
```

Example: Filter and process values lazily
```
evens_squared = (x * x for x in range(10) if x % 2 == 0)
print(list(evens_squared))       #  Output: [0, 4, 16, 36, 64]
```

Example: Nested generator expressions
Nested generator expressions combine multiple iterables into a single, memory-efficient iterator
```
nested = (x * y for x in range(3) for y in range(3, 6))
print(list(nested))                 #  Output: [0, 0, 0, 3, 4, 5, 6, 8, 10]
```

Generators and exceptions
Generators can handle exceptions, ensuring proper cleanup and enabling fine-grained control.

Example: Using yield in try/finally:
```
def safe_generator():
    try:
        yield "Working"
    finally:
        print("Cleanup")

g = safe_generator()
print(next(g))
g.close()                           # Output:   Working...  Cleanup...
```

```
###  Exception injection with throw
def generator_with_error():
    try:
        yield "Before exception"
    except ValueError:
        yield "Handled exception"
g = generator_with_error()
print(next(g))
print(g.throw(ValueError))
                                    # Output:   Before exception... Handled exception
```

Example: Graceful termination with close()
The close() method terminates a generator and ensures cleanup of resources

```
def managed_gen():
    try:
        yield "Start..."
    finally:
        print("Cleanup complete")

g = managed_gen()
print(next(g))                      # Output: Start...
g.close()                           # Output: Cleanup complete
```

Example: Using __del__ for cleanup
If a generator is embedded in a class, __del__ can finalize the generator when the instance is garbage-collected

```
class FinalizedGenerator:
    def __init__(self):
        self.gen = self._generator()

    def _generator(self):
        try:
            yield "Running"
        finally:
            print("Finalized and cleaned up")

    def __del__(self):
        self.gen.close()

gen_instance = FinalizedGenerator()
print(next(gen_instance.gen))  # Output: Running
                                  # Cleanup occurs when gen_instance is garbage-collected.
```

On StopIteration
Left out of this is StopIteration, which is raised automatically when a generator is exhausted, signals the end of iteration. Explicit handling is rarely required in Python.

List Built-in Functions

The following functions apply to all sequences, including tuples and strings.

len(object) - Return the number of items of a sequence or mapping.
max(list) and min(list) - Return the largest or smallest item in the sequence.
any(tuple) - Return True if there is an item in the sequence which is True.
all(tuple) - Return True if all items in the sequence are True.
range([start], stop [, step]) - Works the same as the [ ] operator. Start and step are optional.

List Methods

Unless immutable (like tuples), most of these work with the other object-container types

P.append(object)	Appending one object to end of P
P.extend(list)	Appending multiple lists elements to end of P
P.insert(index, object)	Insert before position index
P.pop(index)	Remove and return item at index (default last, -1) An empty list returns exception
P.remove(value)	Remove first occurrence of value from P. If not in the list returns exception
P.clear()	Removes all elements from the list, leaving it empty.
P.copy()	Creates a shallow copy of list, duplicating elements but not references for nested structures
P.reverse()	Reverse the items in list. Done "in place" does not create new list (get copy of result)
P.sort (key=function)	Custom sorting-use reverse=true to order descending Done "in place" does NOT create new list (make a copy of result)
Accessor methods	
P.count (value)	Return number of occurrences of value in list P
P.index (value)	Return index of first occurrence of value in list P

Simulating List Methods on Tuples

Append, extend, remove, pop aren't applicable to tuples, since tuples are designed to be immutable.
If you need to perform similar functions, here are some work-arounds:

To append tuple with an object, a trailing comma makes object a tuple, to be concatenated (same way you extend the tuple):

```
>>> n_tuple=(1,2,3,4)
>>> n_tuple + (5)
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    n_tuple + (5)
TypeError: can only concatenate tuple (not "int") to tuple
>>> n_tuple + (5,)
(1, 2, 3, 4, 5)
```

Remove and pop: You can't subtract from a tuple but you can update it with a slice of itself:

```
>>> n_tuple - (5)
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    n_tuple - (5)
TypeError: unsupported operand type(s) for -: 'tuple' and 'int'
>>> n_tuple = n_tuple[:4]
>>> n_tuple
(1, 2, 3, 4)
```

Set methods

Since sets aren't ordered, methods mentioned above act slightly different

set1.pop()	Remove an arbitrary object, returning it. If empty, raises a KeyError exception.
set1.add(new)	Adds element new to the set. If the object is already in the set, nothing happens.
set1.remove(old)	Removes element 'old'. If not in the set, will raise a KeyError exception.
set1.discard(old)	Removes element 'old'. If not in the set, nothing happens
set1.clear()	Remove all items from the set.
set1.frozenset()	Creates an immutable version of a set
set1.copy()	Copy set to make a shallow copy. Objects in new set reference same objects in original
set1.update(new)	Merge values, new set into original set, adding elements as needed. Like set1 |= new
set1.intersection_update	Update set1 to have the intersection of set1 and new. Discards elements from set1, keeping only those common to new and set1. It is equivalent to typing set1 &= new.
set1.isdisjoint()	Returns True if two sets have no elements in common (i.e., they are disjointed)
set1.difference_update	Update set1 to have the difference of set1 and new. In effect, this discards elements from set1 which are also in new . It is equivalent to set1 -= new.
set1.symmetric_difference_update	Update set1, symmetric difference between set1 and new. Discards elements from set1 which are common with new and also inserts elements into set1 which are unique to new . It is equivalent to set1 ^= new.

Accessor methods	
set1.issubset(set)	If set1 is a subset of set , return True, otherwise False. Essentially, this is set1 <= set.
set1.issuperset(set)	If set1 is a superset of set , return True, otherwise False. Essentially, this is set1 >= set
set1.union(new) >>> prime.union((1, 2, 3, 4, 5)) set([1, 2, 3, 4, 5, 7, 11, 13])	Return set1 | new. If new is a sequence or other iterable, make a new set from the value of new, then return the union, set1 | new. This does not update set1
set1.intersection	Return string1 & new. If new is a sequence or other iterable, make new set from the value of new, then return the intersection, set1 & new. Does not update set1
set1.difference	Return set1 - new. If new is a sequence or other iterable, make a new set from the value of new , then return the difference, set1 - new. This does not update set1
set1.symmetric_difference	Return set1 ^ new. If new is a sequence or other iterable, make new set from the value of new, then return the symmetric difference, set1 ^ new. Does not update set1

Dictionary Built-in Functions

len(object) - Return the number of items of a sequence or mapping.

dict(valList) - Each element of the list must be a 2-element tuple; create a dict with the first element as the key and the second element as the value. Note that the zip function produces a list of 2-tuples from two parallel lists.

```
>>> dict( [('first',0), ('second',1), ('third',2)] )
  {'second': 1, 'third': 2, 'first': 0}
>>> dict( zip(['fourth','fifth','sixth'],[3,4,5]) )
  {'sixth': 5, 'fifth': 4, 'fourth': 3}
```

Dictionary Methods

dict1.clear()	Remove all items from the dict.
dict1.copy()	Copy to make a new dict. Shallow copy: all references to same objects in original
dict1.setdefault (key [, default])	Get value of key, but also sets a default value
dict1.update(new [, default])	Merge values from new into the original, adding or replacing as needed. It is equivalent to "for k in new.keys(): d[k]= new[k]"
dict1.pop(key [, value])	Remove key, returning the associated value. If the key does not exist, return the *optional value* provided in the pop call.
dict1.get(key [, default])	Get item with key, like dict1[key]. If key is not present, supply default instead.
dict1.has_key(key)	Obsolete in Python 3.x. Use key in dict1 instead
dict1.items()	Return all items in the dict as a sequence of (key,value) tuples in no particular order
dict1.keys()	Return all of the keys in the dict as a sequence of keys (in no particular order)
dict1.values()	Return all the values from the dict as a sequence (returned in no particular order)

String Built-in Functions

chr(i) - Return a string of one character with ordinal i; 0 ≤ i < 256.

len(object) - Return the number of items of a sequence or mapping.

ord(c) - Return the integer ordinal of a one character string

str(object) - Return string representation of the object. If it IS a string, the return value is the same object.

unichr(i) and unicode(string [, encoding,] [errors]) These have been deprecated- use chr() instead

repr(object) - Returns a string representation of an object suitable for debugging, including escape sequences and types when applicable. Can be used within f-strings to display the debug representation of variables, for example: f'{variable!r}'.

String transformation functions (create a new string from an existing string).

string1.capitalize()	Make copy with first character capitalized.
string1.center(width)	Make copy of the string centered in a string of length 'width' padded with spaces.
string1.encode(encoding [, errors])	Return an encoded string version of string1. Default encoding is the current default string encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a ValueError. Other possible values are 'ignore' and 'replace'.
string1.expandtabs([tabsize])	Return a copy of string1 with tab expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.
string1.join(sequence)	Return a string which is the concatenation of the strings in the sequence . The separator between elements is string1
string1.ljust(width)	Return string1 left justified in a string of length width. Padding is done using spaces.
string1.lower()	Return a copy of string1 in all lowercase.
string1.lstrip()	Return a copy of string1 with leading whitespace removed.
string1.replace(old, new [, maxsplit])	Return a copy of string1 with all occurrences of substring old replaced by new. If the optional argument maxsplit is given, only the first maxsplit occurrences are replaced.
string1.rjust(width)	Return string1 right justified in a string of length width . Padding is done using spaces.
string1.rstrip()	Return a copy of string1 with trailing whitespace removed.
string1.strip()	Return a copy of string1 with leading and trailing whitespace removed.
string1.swapcase()	Return a copy with uppercase characters switched to lowercase and vice versa.
string1.title()	Return a titlecased version of string1.

string1.translate() / str.maketrans()	str.maketrans() creates a translation table for str.translate() to map or remove characters.Example: "abc".translate(str.maketrans("a", "b", "c")) replaces a with b and removes c.
string1.upper()	Return a copy of string1 all uppercase.
string1.count(sub [, start] [, end])	Return the number of occurrences of substring sub in string1[start : end]. Optional arguments start and end are interpreted as in slice notation.
string1.endswith(suffix [, start] [, end])	Return True if string1 ends with the specified suffix , otherwise return False. With optional start, or end, test string1[start : end]. The suffix can be a single string or a sequence of individual strings.
string1.startswith(prefix [, start] [, end])	Return True if string1 starts with the specified prefix , otherwise return False. With optional start, or end, test string1[start : end]. The prefix can be a single string or a sequence of individual strings.
string1.find(sub [, start] [, end])	Return the lowest index in string1 where substring sub is found, such that sub is contained within string1[start : end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.
string1.index(sub [, start] [, end])	Like string1.find but raise ValueError when the substring is not found.
string1.isalnum()	Return True if all characters in string1 are alphanumeric, False otherwise.
string1.isalpha()	Return True if all characters in string1 are alphabetic, False otherwise.
string1.isdigit()	Return True if all characters in string1 are digits, False otherwise.
string1.islower()	Return True if all characters in string1 are lowercase, False otherwise.
string1.isspace()	Return True if all characters in string1 are whitespace, False otherwise.
string1.istitle()	Return True if string1 is a titlecased string
string1.isupper()	Return True if all characters in string1 are uppercase, False otherwise.
string1.rfind(sub [, start] [, end])	Return the highest index in string1 where substring sub is found, such that sub is contained within string1[start : end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.
string1.rindex(sub [, start] [, end])	Like string1.rfind but raise ValueError when the substring is not found.
string1.zfill()	Pads a string on the left with zeros to achieve a specified width.
casefold()	Returns case-insensitive suitable for Unicode compare (stronger than lower())
isprintable()	Checks all characters are printable
removeprefix(prefix)	Removes the prefix, if present
removesuffix(suffix)	Removes the suffix, if present

These generators create another kind of object, usually a sequence, from a string

string1.partition(sep) *string1.rpartition()*	Return a three-tuple of the text prior to the first occurrence of sep in the string string1, the sep as the delimiter, and the text after the first occurrence of the separator. If the separator doesn't occur, all of the input string is in the first element of the 3-tuple; the other two elements are empty strings. The str.rpartition() method works like partition but starts splitting from the rightmost occurrence of the separator
string1.split(sep[, maxsplit])	Return a list of the words in the string string1, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.
string1.splitlines([keepends])	Return a list of the lines in string1, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

F-Strings and Formatting

I will not be discussing older Python printing methods like % formatting or str.format() since they are increasingly historical.

Basic formatting types

s	String format	Default for strings	%	Percentage	Multiplies by 100, w/ percent sign
d	Decimal Int	Comma as number separator	x or X	Hexadecimal	Hexadecimal in lowercase or upper
n	Number	Same as d but use locale setting	c	Character	(self-explanatory)
e	Exponent	Scientific notation using letter ‘e’	b	Binary	(self-explanatory)
f	Fixed-point	Fixed-point; default precision is 6	o	Octal	(self-explanatory)

################ General rules on usage, etc.

Using a colon (:) is always required in f-strings when you want to apply any kind of formatting specifier, including width, precision and alignment specifiers discussed below.

Also found in the following examples is the way f-strings behave with quotes:
Inside of print(), double quotes are usually prefered but single also works. Whichever you pick, that type of quote will need to be escaped if it appears inside the f-string (like in f"my cat said \"meow\" loudly" or f'my dog\'s hair is brown')

Backslashes \ generally need escaped inside of any f-string blocks unless part of an escape sequence like \n or \t.
Sometimes, if a project's norm is to use one style of f-strings, you may encounter some text that so many quotes inside that using an alternative f-string style can reduce the need for escaping, making it more efficient for that instance.

Outside print(), single and double quotes can also be used, following the same rules for escaping matching quotes and backslashes. Blocks like f''' (triple single- or triple double-quotes) are for multiline string literals, and quotes won't need escaping (although backslashes will).

```
import math
variable = math.pi
##########  Alignment and spacing (25 chars)  ##
print(f"|{variable:25}|")
print(f"|{variable:<25}|")
print(f"|{variable:>25}|")
print(f"|{variable:^25}|\n")
### Outputs:
|                3.141592653589793        |
| 3.141592653589793                       |
|                3.141592653589793        |
|           3.141592653589793             |

#####  Using a fill character  #####
print(f"|{variable:=<25}|")
print(f"|{variable:=>25}|")
print(f"|{variable:=^25}|\n")
### Outputs:
|3.141592653589793========|
|========3.141592653589793|
|====3.141592653589793====|

#####  Percentage formatting  #####
variable = 4
print(f"This prints with percent formatting {variable:%}\n")
### Outputs:  (use .#% for decimal precision)
This prints with percent formatting 400.000000%

#####  Exponent formatting  ##
variable = 403267890
print(f"This prints with exponential formatting {variable:e}")
### Outputs:
This prints with exponential formatting 4.032679e+08

#####  Adding +/- signing  ##
variable = 1200356.8796
print(f"With a forced plus sign: {variable:+.2f}")
variable *= -1     ## Switch sign
print(f"Signing, two decimal places, commas: {variable:,.2f}")
### Outputs:
With a forced plus sign: +1200356.88
Signing, two decimal places, commas: -1,200,356.88
```

```
#####  Decimals  ##
variable = 1200356.8796
print(f"With two decimal places: {variable:.2f}")
print(f"With three decimal places: {variable:.3f}")
print(f"With two decimal places and a comma: {variable:,.2f}")
### Outputs
With two decimal places: 1200356.88
With three decimal places: 1200356.880
With two decimal places and a comma: 1,200,356.88

#####  Tabular Data with Fixed Widths  ##
print(f'Number\t\tSquare\t\t\tCube')
for x in range(1, 11):
    x = float(x)
    print(f'{x:5.2f}\t\t{x*x:6.2f}\t\t{x*x*x:8.2f}')

###  5, 6, and 8 character field widths, padded w/ spaces

## Printed output below is approximated.
Number  Square   Cube
  1.00      1.00     1.00
  2.00      4.00     8.00

#####  Expressions and Inline Calculations ##
These look slightly different - they are as run in a Python shell
aka interactive interpreter which is available in IDLE and other
IDEs

f"'sum: {some_dict['a'] + some_dict['b']}'"
'sum: 579'
##
>>> f'if statement: {a if a > b else b}'
'if statement: 456'
##
>>> f'min: {min(a, b)}'
'min: 123'
##
>>> f'Hi {username}. And in upper: {username.upper()}'
'Hi doug. And in upper: DOUG'
##
>>> f"Squares: {[x ** 2 for x in range(5)]}"
'Squares: [0, 1, 4, 9, 16]'
```

Using the str.format() method and string functions
Most of the focus on modern Python printing is on f-strings, considering their ease in usage, but the traditional string methods still play a big part of doing more with text- specifically string objects

In automatic positional, {} brackets go in order
    He knows his {}-{}-{}s.format('A', 'B', 'C') "He knows his A-B-Cs"
Numbered positional arguments :
    He knows his {2}-{0}-{1}s.format('B, 'C', 'A') "He knows his A-B-Cs"
Named arguments, defined inline
    He knows his {Z}-{Y}-{X}s'.format(Y='B',Z='A',X='C') "He knows his A-B-Cs"
Indexing within Sequences
    {0[2]}-{0[1]}-{0[0]}s and {1[1]}-{1[2]}-{1[0]}s.format(('C', 'B', 'A'), (3, 1, 2)) "A-B-Cs and 1-2-3s"

Attributes of objects
Real: {.real}, Imag: {a.imag}'.format(1+2j, a=3+4j) "Real: 1.0, Imag: 4.0"
(Note: automatic and numbered fields cannot mix; there are no programmatic issues with other ways (e.g. named))

Using format listing attributes of objects with inline functions:
String Example:
Item: {0}, Data type: {1}, Length: {2}".format("Walnut", type("Walnut").__name__, len("Walnut"))
Output: "Item: Walnut, Data type: str, Length: 6"
List Example:
Item: {0}, Data type: {1}, Length: {2}".format(lst := ["Apple", "Banana", "Cherry"],type(lst).__name__, len(lst))
Output: "Item: ['Apple', 'Banana', 'Cherry'], Data type: list, Length: 3"
Specific list element example
Item: {0}, Data type: {1}, Length: {2}".format(item := lst[2],type(item).__name__,len(item))
Output: "Item: Cherry, Data type: str, Length: 6"
Dictionary Example:
Item: {0}, Data type: {1}, Length: {2}".format(dct := {'Name': 'Alice', 'Age': 30, 'City': 'London'},type(dct).__name__,len(dct))
Output: "Item: {'Name': 'Alice', 'Age': 30, 'City': 'London'}, Data type: dict, Length: 3"
Specific dictionary value example
Item: {0}, Data type: {1}, Length: {2}".format(value := dct['City'],type(value).__name__,len(value))
Output: "Item: London, Data type: str, Length: 6"

External variable definitions v.s. inline definitions
Using print(sentence) for the output, we get this for both of the examples below: "In 2005, Akira Haraguchi recited the number pi to 112 digits during a Pi Day celebration that also featured puzzles."
External variables:
```
dict_data = {"digits": 112}
str_data = "Akira Haraguchi"
int_data = 2005
list_data = ["Pi Day celebration", "puzzles"]

# Using str.format() to format the sentence
sentence = "In {0}, {1} recited the number pi to {2[digits]} digits during a {3[0]} that also featured {3[1]}.".format(
    int_data, str_data, dict_data, list_data
)
```
Inline definitions:
```
sentence = "In {0}, {1} recited the number pi to {2[digits]} digits during a {3[0]} that also featured {3[1]}.".format(
    2005, "Akira Haraguchi", {"digits": 112}, ["Pi Day celebration", "puzzles"]
)
```

Value Conversion
!r for repr(): Converts the value to its repr() representation, typically for debugging.
```
value1 = "Hello\tWorld"
print("Normal: {}".format(value1))     # Hello    World (tab invisible)
print("Repr: {!r}".format(value1))     # 'Hello\tWorld' (tab revealed)

value2 = "    "  # Four spaces
print("Normal: {}".format(value2))     # (Looks blank)
print("Repr: {!r}".format(value2))     # '    ' (Shows spaces explicitly)
```

!a for ascii(): Converts the value using ascii(), escaping non-ASCII characters.
```
value1 = "Hello, 世界!"  # Includes non-ASCII characters
print("Normal: {}".format(value1))  # Hello, 世界!
print("ASCII: {!a}".format(value1))  # 'Hello, \u4e16\u754c!'

value2 = "Café"         # Includes a non-ASCII accented character
print("Normal: {}".format(value2))  # Café
print("ASCII: {!a}".format(value2))  # 'Caf\u00e9'
```

!s for str(): Is primarily useful when an object has a __repr__() but no __str__(), or when seeking clues in debugging.

Value Formatting
{[selector][conversion]:[format_specifier]}.format(value)
Format Specifier Options:

Fill and Alignment:
{: <10}.format('left')   # Left-aligned, padded
{:^10}.format('center') # Center-aligned
{:*>10}.format('right') # Right-aligned, padded with '*'

Sign: Show signs for numbers (+, -, or space for positive).
{:+}.format(42) # +42
{: }.format(-42) # -42

Width and Precision:
{:8.2f}.format(3.14159) # Width 8, 2 decimal places

Comma for Thousands:
{:,}.format(1234567) # 1,234,567

Type: Control numeric or string types:
{:b}.format(42)   # Binary (101010)
{:x}.format(255) # Hexadecimal (ff)
{:e}.format(1234) # Exponential (1.234e+03)

Functions
Functions are first-class objects- can be assigned to variables, passed as arguments, and returned from other functions.
Methods are functions bound to a class or an instance.
- Standalone Functions: Defined using the 'def' keyword.
- Instance Methods: Bound to an instance of a class; require 'self' as the first parameter.
- Class Methods: Bound to a class; require 'cls' as the first parameter. Declared with '@classmethod'.
- Static Methods: Need the class namespace to exist since they are defined inside, but are otherwise like instance methods.

```
# Standalone functions are defined outside the class - independent and not tied to a class or instance
    def standalone_function():
        return "Standalone function called"

# Instance method - default method type inside a class, bound to the instance
    class Example:
        def instance_method(self):
            return f"Instance method called on {self}"

# Class method (bound to the class, not an instance)
    @classmethod
    def class_method(cls):
        return f"Class method called on {cls}"

# Static methods have independent logic and are like standalone functions, but their function definition is in the class
    @staticmethod
        def static_method():
            return "Static method called"
```

Usage and further notes

```
    print(standalone_function())    # Outputs: Standalone function called

# Using the Example class - making an instance to demonstrate the following method types
    obj = Example()

# Instance method (requires an instance to call it)
    print(obj.instance_method())   # Outputs: Instance method called on <__main__.Example object at 0x...>
    # Automatically receives the instance (self) as its first argument.
    # Can access instance-specific data or call other instance methods

# Class method (typically called on the class, but can also be called on an instance)
    print(Example.class_method())          # Outputs: Class method called on <class '__main__.Example'>
    print(obj.class_method())              # Outputs: Class method called on <class '__main__.Example'>
    # Automatically receives the class (cls) as its first argument.
    # Often for a method to operate on the class itself (to create alternate constructors, modifying class attributes)
```

```
# Static method (can be called on from both the instance and the class)
    print(obj.static_method())              # Outputs (from instance): Static method called
    print(Example.static_method())          # Outputs (from class): Static method called
    # Does not receive self or cls. It's basically a standalone function but the class must exist for it to be defined
```

[Static and class methods use @classmethod and@staticmethod when they are declared- These are examples of what are called decorators which are very useful and have their own section later.]

Local Functions

Local functions are functions defined within another function.
- recreated with each execution of the enclosing function, resulting in unique objects.
- enclosing scope includes the parameters and local variables of the enclosing function.
- useful for encapsulating logic that doesn't need to be accessed outside the enclosing function.

```
    def outer_function():
        def inner_function():
            return "Hello from inner_function!"
        return inner_function
    func1 = outer_function()
    func2 = outer_function()
    print(func1 is func2)  # False
```

Higher-order functions

Recall that functions are first class objects meaning they can be assigned to variables, passed as arguments, and returned by other functions. A higher-order function either takes a function as an argument or returns one.
Examples: map(), filter(), reduce().

```
    def square(x):
        return x * x
    print(list(map(square, [1, 2, 3])))         # Output: [1, 4, 9]
```

Iterators

An iterator is an object that represents a stream of data. It implements the __iter__() and __next__() methods.
    __iter__(): Returns the iterator object itself.
    __next__(): Returns the next value in the sequence or raises a StopIteration exception when the sequence ends.

Using iter() on iterable containers (like lists, tuples, dictionaries) you can get an iterator object.

```
    numbers = [1, 2, 3]
    iterator = iter(numbers)
    print(next(iterator))    # outputs 1
    print(next(iterator))    # outputs 2
```

You can define your own iterator by implementing the __iter__() and __next__() methods.

```
    class Countdown:
        def __init__(self, start):
            self.current = start

        def __iter__(self):
            return self

        def __next__(self):
            if self.current <= 0:
            raise StopIteration
            self.current -= 1
            return self.current + 1

    for num in Countdown(5):
    print(num)  # Outputs: 5, 4, 3, 2, 1
```

Use Cases:
    Efficient looping through large datasets without loading all data into memory.
    Lazy evaluation in scenarios where data generation is costly.

Recursive functions

Base case: Prevents infinite recursion by defining a condition where the function stops calling itself.
Recursive step: Progresses towards the base case, reducing the problem size at each step.

```
def factorial(n: int) -> int:
    """Computes factorial using recursion."""
    if n == 0:  # Base case
    return 1
return n * factorial(n - 1)  # Recursive step
print(factorial(5))  # Outputs: 120
```

Encapsulation in Python

Python does not enforce strict access control (like private, protected, or public). Instead, it relies on conventions:
Single underscore (_varname): Indicates a variable or method is intended for internal use only but remains accessible.
Double underscore (__varname): Triggers name-mangling to make the variable harder to access from outside its class.

- Avoid using global variables as they make debugging and maintaining the code harder.
- Prefer passing values as function arguments and returning results instead of modifying external state.

Mutable default arguments in functions

Problem: Mutable default arguments can lead to unexpected behavior:

```
def add_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list
add_to_list(1)  # [1]
add_to_list(2)  # [1, 2] (unintended)
```

Solution: Use None as the default value and initialize inside the function:

```
def add_to_list(value, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(value)
    return my_list
```

Extended formal argument syntax

Variable-length arguments:

```
def func(a, *args, **kwargs):
    print(a, args, kwargs)
func(1, 2, 3, key="value")          # Output: 1 (2, 3) {'key': 'value'}
```

Keyword-only arguments:

```
def func(a, *, b):
    print(a, b)
func(1, b=2)
```

Default argument values:

```
def greet(name="World"):
    return f"Hello, {name}!"
print(greet())  # Hello, World
print(greet("Alice"))  # Hello, Alice
```

Positional-only parameters: parameters behind the "/" must be passed by position- not with keywords

```
def add(x, y, /, z):
    return x + y + z
add(1, 2, z=3)          # Valid
add(x=1, y=2, z=3)      # Error: 'x' and 'y' are positional-only
```

The packages mypy and pydantic (both available through pip) can provide more explanatory type errors. Rather than simply "Type mismatch", it would specify "Argument 2 to 'add' has incompatible type 'str'; expected 'int' "

Error Handling in Functions

Using try/except: Catch specific exceptions to handle errors gracefully.

```
def read_file(filepath: str) -> str:
    try:
        with open(filepath, 'r') as file:
          return file.read()
    except FileNotFoundError:
        return "File not found."
```

Raising exceptions when function inputs are invalid or an operation cannot be performed.

```
def square_root(x: float) -> float:
    if x < 0:
        raise ValueError("Cannot compute square root of a negative number.")
    return x ** 0.5
```

Function introspection and performance

dir(): Lists attributes and methods of an object.

```
def sample_func():
    pass

print(dir(sample_func))  # Lists methods like __call__, __doc__, etc.
```

help(): Provides docstrings embedded in objects

```
help(sample_func)
```

inspect(): examine functions in depth

```
import inspect
def sample_func(a, b=10):
    return a + b

print(inspect.signature(sample_func))  # (a, b=10)
print(inspect.getsource(sample_func))  # Source code of the function
```

timeit(): to measure function execution time

```
import timeit
def slow_function():
    result = 0
    for i in range(1000):
        result += i
    return result

print(timeit.timeit(slow_function, number=1000))
```

Custom function attributes, allow storing metadata, configurations, or state

```
def sample_function():
   pass

    sample_function.author = "Alice"
    sample_function.version = 1.0
    sample_function.default_value = 42
    sample_function.saved_state = {"last_run": None}
# above are default values

print(sample_function.author)            # Outputs: Alice
print(sample_function.version)           # Outputs: 1.0
print(sample_function.default_value)     # Outputs: 100
print(sample_function.saved_state)       # Outputs: {'last_run': '2023-11-03'}
```

Coroutines and Async

Coroutines, defined using async def, enable non-blocking operations, allowing tasks like I/O or network calls to run efficiently without halting other processes

```
import asyncio
async def greet():
   print("Hello")
   await asyncio.sleep(1)       # Non-blocking pause
   print("World")
asyncio.run(greet())            # Outputs: Hello (pauses), World
```

- async marks a function as a coroutine.
- await pauses execution until the awaited task completes.
- Ideal for handling concurrent operations such as web requests or file I/O.

Dynamic Function Creation

Example 1: Using exec()

```
code = "def dynamic_func(x): return x * 2"    # Define a function dynamically as a string
namespace = {}                                # Create a namespace dictionary to store the function
exec(code, namespace)                         # Execute the code string in the given namespace
dynamic_func = namespace["dynamic_func"]          # Retrieve the dynamically created function from the namespace

    # Call the function with a value for x
result = dynamic_func(5)              # Here, x is 5
print(result)                         # Outputs: 10
```

- String code definition defines a function, dynamic_func, which multiplies its argument x by 2.
- The namespace dictionary acts as a container where exec() will place the dynamically created dynamic_func.
- Execution: exec(code, namespace) evaluates the string code and stores the resulting dynamic_func in the namespace.
- The namespace["dynamic_func"] retrieves the dynamic_func function object from the namespace.
- Calling the function: dynamic_func(5) executes the function, passing 5 as x. The result, 5 * 2 = 10, is stored in result.

Example 2: Using types.FunctionType

```
import types
func_code = """               # Define a function body as a string
def dynamic_func(x):
   return x * 3
"""
code_obj = compile(func_code, "<string>", "exec")    # Compile the function code into a code object
namespace = {}               # Create a namespace to store the function
exec(code_obj, namespace)         # Execute the compiled code within the namespace

dynamic_func = types.FunctionType(                  # Create a function using types.FunctionType explicitly
    code=namespace["dynamic_func"].__code__,  # Function code object
    globals=namespace,                        # Globals for the function
    name="dynamic_func"                       # Optional: explicitly set the function's name
    )

result = dynamic_func(4)              # Call the function
print(result)                         # Outputs: 12
```

Explanation of Code

- types.FunctionType: Creates a function from a code object and a namespace (for global variables).
  Offers explicit control over the construction of the function object.
- compile(): Converts a string of Python code into a code object, which can be executed or used to create a function.
- exec(): Executes the code object in the specified namespace.
- This method is more structured and allows finer control than directly using exec() on raw strings.

Best Practices for Function Design

- Small, focused functions- should perform a single, well-defined task.

```
def calculate_area(length: float, width: float) -> float:
    return length * width
```

- Typing and annotations

Improve code readability and allow static type checkers to identify type errors.
Available in the typing module.

```
from typing import List, Tuple
def get_coordinates() -> Tuple[float, float]:
    return 40.7128, -74.0060

def greet_all(names: List[str]) -> None:
    for name in names:
        print(f"Hello, {name}")
```

- Documenting functions: use docstrings

```
def divide(a: float, b: float) -> float:
"""
Divides two numbers: a (float) numerator, and b (float) denominator
Returns: (float) result of division
"""
if b == 0:
    raise ValueError("Denominator cannot be zero.")
return a / b
```

Classes

```
### Defining a Class ###
class Shape:
    ### Defining Attributes and Methods
    def __init__(self, color="black"):
            """Initializer for the Shape class."""
        self.color = color                  # Public attribute
        self._dimensions = []               # Protected attribute to hold dimensions

    def add_dimension(self, dimension):     # Adds a dimension to the shape (example of instance method)
        self._dimensions.append(dimension)

    def __str__(self):                      # Operator Overloading: Customize how the object is printed."""
        return f"Shape(color={self.color}, dimensions={self._dimensions})"

    @staticmethod
    def describe():                         # Static Method: Does not depend on class or instance state."""
        return "Shapes are objects with geometrical properties."

    @classmethod
    def default_shape(cls):                 # Class Method: Accesses the class itself."""
        return cls(color="blue")

    @property
    def dimensions(self):                   #Getter for dimensions."""
        return self._dimensions

    @dimensions.setter
    def dimensions(self, value):            # Setter for dimensions with validation."""
        if not isinstance(value, list):
            raise ValueError("Dimensions must be a list.")
        self._dimensions = value


### Creating an Instance ###
shape1 = Shape(color="red")
shape1.add_dimension(5)
shape1.add_dimension(10)
print(shape1)
            ### Using Instance and Class Methods
print(Shape.describe())                     # Call to static method
default_shape = Shape.default_shape()       # Call to class method
print(default_shape)
            ### Encapsulation via Public/Private Attributes and Controlled Access
print("Dimensions (getter):", shape1.dimensions)
shape1.dimensions = [15, 20]                # Using the setter
print("Updated dimensions:", shape1.dimensions)
            ### Inheritance and Polymorphism
class Circle(Shape):                        # Circle inherits from Shape
    def __init__(self, radius, color="black"):  # __init__ being the constructor
        super().__init__(color)             # super() means child class is to use this function as defined in parent class
        self.radius = radius

    def area(self):                         #Polymorphic method
        from math import pi
        return pi * self.radius**2

    def __str__(self):                      # Override parent class's string representation
        return f"Circle(color={self.color}, radius={self.radius})"


### Create and use a child class ###
circle = Circle(radius=7, color="green")
print(circle)
print(f"Circle area: {circle.area()}")
```

```
        ###  Demonstrating polymorphism   ###
shapes = [shape1, default_shape, circle]
for shape in shapes:
   print(shape)                    # __str__ is polymorphic across different shapes
```

Key Features and explanations

Defining and Using Classes:

- The Shape class is a foundational blueprint with attributes (color, _dimensions) and methods

Attributes and Methods:

- Attributes: color is public, _dimensions is protected (underscored by convention).
- Methods include:
  - Instance Method: add_dimension modifies specific instance data
  - Static Method: describe gives info about the class concept without accessing instance or class data
  - Class Method: default_shape creates a predefined Shape instance

Operator overloading and polymorphism:

- __str__ is overridden to customize behavior of print() and string representations for Shape and Circle instances.
- Circle overrides the __str__ method to provide its custom string representation.
- The program treats different objects (Shape and Circle) uniformly in a loop. Each object calls its version of the __str__ method, depending on its class:

```
shapes = [shape1, default_shape, circle]
for shape in shapes:
   print(shape)                    # Calls the appropriate __str__ method
```

Encapsulation via Public/Private Attributes and Controlled Access:

- The _dimensions attribute is protected (conventionally indicated with an underscore), discouraging direct access.
- The @property decorator is a controlled way to access with getter (dimensions) and setter (dimensions.setter)

Function Overriding and Overloading in Python

While overriding is supported in Python, overloading works differently compared to languages like Java or C++. Here's an explanation:

Function Overriding

Function overriding occurs when a subclass provides a specific implementation of a method that is already defined in its parent class. The subclass's version overrides the parent class's version.
The method name and signature in the subclass must match the parent class's method.
The overridden method in the subclass will be invoked when called on an instance of the subclass.

```
class Parent:
   def greet(self):
      return "Hello from Parent!"

class Child(Parent):
   def greet(self):
      return "Hello from Child!"

obj = Child()
print(obj.greet())              # Outputs: "Hello from Child!"
```

Function Overloading

Function overloading allows multiple methods with the same name but different parameters (signature). Python does not natively support function overloading. However, this can be achieved using default arguments or *args and **kwargs.
Python uses the latest defined method if multiple methods share the same name.
Functionality similar to overloading can be implemented manually by checking argument types.

Simulating Overloading:

```
class Calculator:
   def add(self, *args):
      if len(args) == 2:
         return args[0] + args[1]
      elif len(args) == 3:
         return args[0] + args[1] + args[2]
      else:
         raise ValueError("Unsupported number of arguments")

calc = Calculator()
print(calc.add(1, 2))       # Outputs: 3
print(calc.add(1, 2, 3))    # Outputs: 6
```

Differences between Overriding and Overloading

	Overriding	Overloading
Definition	Redefines a method in a subclass.	Allows multiple methods with same name, different arguments.
Scope	Happens in inheritance (parent-child).	Happens in the same class or module.
Support in Python	Fully supported.	Not natively supported; simulated with argument handling.
Execution	Depends on object type (runtime).	Depends on argument count or type (design time).

Superclasses and the super() function

A superclass (or parent class) defines attributes and methods that can be inherited by subclasses
The super() function allows a subclass to manage inheritance of methods and attributes
It helps to not need to repeat parent class names over and over again
If you override/overload a function in your subclass, you can use super().method_name() to invoke the original method
Generally super() helps traversing the class hierarchy based on the Method Resolution Order (MRO)
(the MRO determines the order in which classes are searched for methods or attributes)

```
class Animal:
    def sound(self):
        return "Some sound"
class Dog(Animal):
    def sound(self):
        return super().sound() + " and a bark"
print(Dog().sound())   # Output: Some sound and a bark
```

Simplified Inheritance Management:

The super() function allows a subclass to call methods and access attributes from its superclass without explicitly naming it.

```
class Parent:
    def greet(self):
        return "Hello from Parent"
class Child(Parent):
    def greet(self):
        return super().greet() + " and Child"
print(Child().greet())   # Outputs: Hello from Parent and Child
```

Method Resolution Order (MRO):

super() uses the MRO to determine the order in which classes are searched for methods and attributes. This is particularly useful in multiple inheritance scenarios.

```
class A:
    def process(self):
    return "A"
class B(A):
    def process(self):
    return super().process() + " B"
class C(B):
    def process(self):
        return super().process() + " C"
  print(C().process())  # Outputs: A B C
```

Metaclasses

Metaclasses are "classes of classes," defining how classes themselves behave. They control class creation and customization at the metaprogramming level.

- Defining Custom Class Behavior:

By overriding methods like __new__ or __init__, metaclasses can modify or enforce rules for class creation.

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        dct["greet"] = lambda self: f"Hello from {name}"
        return super().__new__(cls, name, bases, dct)
class CustomClass(metaclass=Meta):
    pass

obj = CustomClass()
print(obj.greet()) # Outputs: Hello from CustomClass
```

- Common Use Cases:
    - Enforcing constraints on class definitions.
    - Injecting methods or attributes into classes.
    - Creating frameworks or APIs requiring specialized behavior.

Example: subclassing immutable types
- Metaclasses are often used for creating specialized types, such as subclassing immutable types like tuples.

```
class SortedTuple(tuple):
    def __new__(cls, iterable):
        return super().__new__(cls, sorted(iterable))

sorted_tuple = SortedTuple([3, 1, 2])
print(sorted_tuple)     # Outputs: (1, 2, 3)
```

Example: Enforcing attribute naming rules with a metaclass
- Ensure that all attributes of a class must be uppercase, raising an error if any lowercase attribute is defined.

```
class UppercaseAttributesMeta(type):
    def __new__(cls, name, bases, dct):
        for key in dct:
            if not key.isupper() and not key.startswith("__"):
                raise ValueError(f"Attribute '{key}' is not uppercase in class '{name}'.")
        return super().__new__(cls, name, bases, dct)

# Using the metaclass to enforce uppercase attribute names
class MyClass(metaclass=UppercaseAttributesMeta):
    ATTRIBUTE_1 = "value1"
    ATTRIBUTE_2 = "value2"

# This will raise an error because the attribute 'attribute_3' is not uppercase
try:
    class InvalidClass(metaclass=UppercaseAttributesMeta):
        attribute_3 = "value3"
except ValueError as e:
    print(e)  # Outputs: Attribute 'attribute_3' is not uppercase in class 'InvalidClass'
```

- The UppercaseAttributesMeta metaclass checks all attributes defined in the class dictionary (dct).
- It raises a ValueError if any attribute name is not fully uppercase (excluding special names starting with __).
- This ensures that classes using this metaclass adhere to a specific naming convention, which can be useful in codebases requiring strict attribute naming rules.

LEGB: Understanding Python's Scope Resolution

When Python encounters a variable name, it searches for the variable's definition in the following order:

- Local Scope: The innermost function or block where the variable is defined.
- Enclosing Scope (aka nonlocal): The scope of any enclosing function or lambda, excluding the global scope.
- Global Scope: The top-level scope of the module where the code is written.
- Built-in Scope: The pre-defined names in Python, such as print, len, etc.

If the variable is then still not found, it raises a NameError.

Previously the unrelated Method Resolution Order (MRO) was mentioned, determining the lookup order for methods and attributes in class hierarchy. LEGB instead governs how Python resolves variables or functions based on their scope and visibility.

Built-in Scope print()

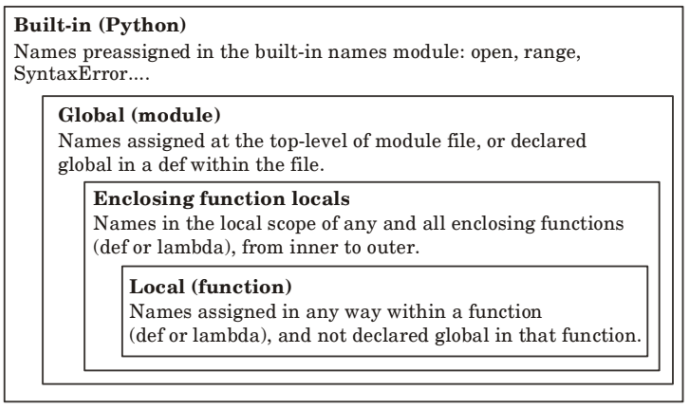
```
x = "Global Scope"

def outer_func():
    x = "Enclosing Scope"

    def inner_func():
        x = "Local Scope"
        print(x)

    inner_func()

outer_func()
```



Detailed Explanation of Each Scope

```
x = 100  # Global scope
def outer_function():
    x = 200  # Local scope of outer_function

    def inner_function1():
        x = 300  # Local scope of inner_function1
        return x  # Refers to 'x' in the local scope of inner_function1

    def inner_function2():
        x = 60  # Local scope of inner_function2
        return x  # Refers to 'x' in the local scope of inner_function2

    return inner_function1(), inner_function2()  # Calling both functions in the enclosing scope
print(outer_function())  # Output: (300, 60)
print(x)  # Output: 200 (Global 'x' is unchanged)
```

Details:

- Global x = 100: The x in the global scope remains unaffected.

- Inside outer_function(), there's a local x = 200. However, this x isn't used in inner_function1() or inner_function2() because both functions define their own local x.
- Inside inner_function1(), there is local x = 300; return x is the output of inner_function1()
- Inside inner_function2(), there is local x = 60; return x is the output of inner_function2()

When outer_function() is called, it runs inner_function1() and inner_function2(), returns a tuple of the two values: (300, 60)

Enclosing gets a bit tricky at first since it is perspective looking out from what is local.
Consider 2 people living in a house with thier own locked rooms and a common area- you have access to the common areas in the (enclosing) house and your room (local) but no access to your roommate's room.

Inside of inner_function1, the enclosing scope is outer_function()'s contents. Even though that includes inner_function2, it does NOT include it's contents. From the view, im enclosed by outer_function(), not inner_function2()
Let's say we get rid of all definitions of x except for outer_function's x=200.
From inside one of the inner_functions, a call for x won't find a local, so it moves onto enclosing, and then finds x=200.
This wouldn't happen if it was only in the other inner_function, since it is STRICTLY refering to the level above its own block.

If it wasn't defined anywhere except right outside of outer_function(), it moves to global scope and finds x=100.
Consider global up to the top-most levels accessible to any function or block when a matching local or enclosing isn't found.

Built-in Scope: Contains Python's built-in functions and constants.
Examples include len, range, and print.

```
# Built-in Scope Example:
def builtin_example():
   print(len("Python"))  # Accesses the built-in len() function
builtin_example()  # Output: 6
```

Lambdas

A lambda function is a small, anonymous function defined using the lambda keyword.
Primarily used for short-lived, single-expression functions where defining a full function is unnecessary.
Ideal for short, "throwaway" functions used within functional programming or sorting and filtering .

```
lambda arguments: expression
```

Basic syntax:

The arguments are inputs to the lambda function, the expression is evaluated and returned as the output.

```
add = lambda x, y: x + y
print(add(2, 3))                    # Outputs: 5
```

Use cases

Inline function definition:
Lambdas are useful for simple functions defined on-the-fly.

```
square = lambda x: x ** 2
print(square(4))                    # Outputs: 16
```

Lambdas with Multiple Arguments:
Lambdas can accept multiple arguments for concise operations.

```
mul = lambda a, b: a * b
result = mul(5, 3)
print(result)  # Outputs: 15
```

Lambda Function with No Arguments:
Lambdas can be defined without arguments, often used for returning constants.

```
six = lambda: 6
result = six()
print(result)  # Outputs: 6
```

Sorting with a key function:
Lambdas can act as the key argument in sorting operations.

```
people = [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}]
sorted_people = sorted(people, key=lambda x: x['age'])
print(sorted_people)  # Outputs: [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}]
```

Higher-order functions:
Lambdas work seamlessly with functional programming tools like map, filter, and reduce.

Using map: Applies a function to all items in an iterable.

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, nums))
print(squared)              # Outputs: [1, 4, 9, 16]
```

Using filter: Filters elements from an iterable based on a condition.

```
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)                # Outputs: [2, 4]
```

Using reduce: Reduces an iterable to a single value by applying a function cumulatively

```
from functools import reduce
nums = [1, 2, 3, 4]
total = reduce(lambda x, y: x + y, nums)
print(total)                # Outputs: 10
```

Recursive expressions with lambdas:

```
factorial = lambda n: n * factorial(n-1) if n > 1 else 1
print(factorial(5))         # Output: 120
```

Returning a Lambda Function from Another Function:
Functions can return lambdas, enabling dynamic behavior.

```
import math
def myfunc(n):
return lambda a: math.pow(a, n)

square = myfunc(2)
cube = myfunc(3)
squareroot = myfunc(0.5)
print(square(3))            # Outputs: 9.0
print(cube(3))              # Outputs: 27.0
print(squareroot(3))        # Outputs: 1.7320508075688772
```

Dynamically create multiple lambda functions:

```
funcs = [lambda x, n=n: x + n for n in range(5)]
for func in funcs:
print(func(10))             # Outputs: 10, 11, 12, 13, 14
```

Combining lambdas with closures:
Lambdas can capture and use variables from their enclosing scope, creating powerful combinations with closures.

```
def multiplier(n):
return lambda x: x * n

doubler = multiplier(2)
print(doubler(5))           # Outputs: 10
```

Limitations:

Lambdas are limited to a single expression, cannot include statements such as if, while, or for.

Conditional logic workaround using ternary operators:

```
max_val = lambda x, y: x if x > y else y
print(max_val(3, 5))        # Outputs: 5
```

Conditional logic workaround 2:

```
categorize = lambda x: 'even' if x % 2 == 0 else 'odd'
print(categorize(4))        # Outputs: 'even'
```

Common mistakes

- Avoid deeply nested or complex lambdas that reduce maintainability
- Ensure readability and reserve complex logic for named functions
- It's harder to identify lambdas in error messages or stack traces, without a name unless assigned to a variable

- Overusing lambdas

Instead of making this: [result = (lambda x: (lambda y: x + y)(5))(10)]

Do this instead:

```
def add_five(x):
    return x + 5
print(add_five(10))         # Outputs: 15
```

- Improper use in loops:

A common mistake:

```
funcs = [lambda x: x + i for i in range(3)]
for func in funcs:
    print(func(10))         # Outputs: 12, 12, 12
```

The correct approach:

```
funcs = [lambda x, i=i: x + i for i in range(3)]
for func in funcs:
    print(func(10))         # Outputs: 10, 11, 12
```

Closures

A function that captures, retains access to variables from its enclosing scope- even if scope is no longer active- enabling encapsulation and dynamic behavior. To be called a closure, the code block must have a nested function that refers to a value defined in the enclosing function, which then returns the nested function

Basic syntax:

```
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

closure_func = outer_function("Hello, World!")
closure_func()  # Outputs: Hello, World!
```

Practical use cases for closures

Closures can encapsulate variables and logic, providing a lightweight alternative to objects for maintaining state.

Maintaining state across function calls:

```
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

incrementer = counter()
print(incrementer())        # Outputs: 1
print(incrementer())        # Outputs: 2
```

Dynamic function creation (closure version of a 'function factory'):

```
def multiplier(factor):
    def multiply_by(x):     #  A simplified version is shown later in the section on lambdas: [ return lambda x: x * n ]
        return x * factor
    return multiply_by
```

```
double = multiplier(2)
triple = multiplier(3)
print(double(5))              # Outputs: 10
print(triple(5))              # Outputs: 15
```

Decorators often leverage closures to capture and retain context, such as parameters or state, allowing them to dynamically modify or extend function behavior.

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet():
    print("Hello!")

greet()
```

Decorators

Decorators are callable objects (function, class, or instance) that takes another callable as input and returns a new callable, often wrapping or modifying its behavior.
They often rely on closures, which enable them to dynamically change or extend function behavior

- Increase clarity, maintainability, and reducing complexity.
- Manage function calls and instances; for classes, enable customizing object behavior or adding stateful logic
- Augmenting functionality, such as logging, validation, or data transformation
- Proxy calls to intercept, monitor, or modify behavior dynamically

Basic syntax:

```
message = "This is a message."
def decorator(func):
    def modify_message():
        return func().upper()
    return modify_message

def display_message():
    return message                  # Undecorated function

@decorator
def decorated_display_message():
    return message  # Decorated function

print(display_message())                    # Undecorated output, # This is a message
print(decorated_display_message())          # Decorated output,  # Decorated output
```

Prepending the function definition with the @ symbol (as above) to automatically apply the decorator is most common. However, decorators can also be applied manually in the form decorator_name(function_name)() if needed.

Built-in decorators:

@staticmethod and @classmethod are built-in decorators provided by Python for working with methods inside a class. These decorators do not involve replacing the function with an instance of a class, unlike when you use a custom class as a decorator. For the property built-in decorator, (a function) there is a small section reserved for it after this section.

@staticmethod: Defines a method that doesn't rely on instance or class-level data.

```
class Utility:
    @staticmethod
    def add(a, b):
        return a + b
```

@classmethod: Operates at the class level, taking 'cls' as the first parameter

```
    value = 42
    @classmethod
    def get_value(cls):
        return cls.value
```

@property: Allows a method to be accessed as an attribute.

Practical decorators:

Logging:

```
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} and {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log
def add(a, b):
    return a + b

print(add(3, 4))
```

Parameter validation:

```
def smart_divide(func):
        def inner(a, b):
            if b == 0:
                print("Whoops! cannot divide")
                return
            return func(a, b)
        return inner
@smart_divide
def divide(a, b): return a / b
 divide(2, 0)  # Output: Whoops! cannot divide
```

Timing:

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)
    print("Finished!")
slow_function()
```

Classes as decorators

Decorating a function with a class turns the function into an instance of the class.
The decorator class receives the function as an argument to its constructor (__init__) and must implement a __call__ method to enable callable behavior.

```
class CallCount:
    def __init__(self, f):
        self.f = f
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        return self.f(*args, **kwargs)

@CallCount
def hello(name):
    print(f"Hello, {name}!")

hello("Ike")          # Hello, Ike
hello("Selma")     # Hello, Selma
print(hello.count) # Output: 2
```

Instances as Decorators
Instances of classes can act as decorators, enabling dynamic control over their behavior.
Example: tracing calls, allows toggling its functionality (self.enabled),

```
class Trace:
    def __init__(self):
        self.enabled = True

    def __call__(self, f):
        def wrap(*args, **kwargs):
            if self.enabled:
                print(f"Calling {f.__name__}")
            return f(*args, **kwargs)
        return wrap

tracer = Trace()

@tracer
def rotate_list(l):
    return l[1:] + [l[0]]

l = [1, 2, 3]
print(rotate_list(l))           # Calling rotate_list; Output: [2, 3, 1]
tracer.enabled = False
print(rotate_list(l))           # No call trace; Output: [3, 1, 2]
```

Functions as Decorators
Example: unicode escaping- ensures all return values have non-ASCII characters converted to escape sequences

```
def escape_unicode(f):
    def wrap(*args, **kwargs):          #  wrap() closure retains access to 'f' even after escape_unicode returns
        x = f(*args, **kwargs)
        return ascii(x)
    return wrap

@escape_unicode
def a_city():
    return 'São Paulo'

print(a_city())       # Output: 'S\\xe3o Paulo'
```

Functions decorating a class
Decorating classes with a function can enforce design patterns like singletons, dynamically add attributes, or standardize behavior across multiple classes without altering their definitions.

A singleton ensures that only one instance of a class is created, and it provides a global point of access to that instance. This example ensures the same instance is returned every time the class is called, thus behaving as a singleton.

```
def singleton(cls):
    instances = {}  # Dictionary to store a single instance of the class
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]
    return get_instance

@singleton
class DatabaseConnection:
    pass

db1 = DatabaseConnection()
db2 = DatabaseConnection()
print(db1 is db2)  # Outputs: True
```

Chaining multiple decorators:
When stacking decorators, the order of execution is bottom to top (inner to outer).

```
@decorator1
@decorator2
@decorator3
def some_function():
    pass
```

Equivalent to: some_function = decorator1(decorator2(decorator3(some_function)))

Preserving metadata using functools.wraps
Naive decorators can overwrite original metadata (function names, docstrings, etc) functools.wraps can preserve them.

```
import functools

def my_decorator(f):
    @functools.wraps(f)
    def wrap(*args, **kwargs):
        return f(*args, **kwargs)
    return wrap
```

Decorator factories
A decorator factory is a function that returns a decorator, enabling parameterized behavior.

Example: argument validation

```
def check_non_negative(index):
    def validator(f):
        def wrap(*args):
            if args[index] < 0:
                raise ValueError(f"Argument {index} must be non-negative")
            return f(*args)
        return wrap
    return validator

@check_non_negative(1)
def create_list(value, size):
    return [value] * size

print(create_list(1, 3))        # Output: [1, 1, 1]
create_list(1, -3)              # Raises ValueError
```

check_non_negative is a factory that generates the actual decorator (validator).

Property() decorator and Descriptors
Descriptors and the @property decorator are features designed for managing attribute access and behavior. They overlap in functionality, with both capable of controlling how attributes are accessed, modified, or deleted.
@property is ideal for simple, class-specific logic, while descriptors excel in reuse and state management. By understanding their strengths, you can choose the best approach for your specific needs.

Comparison: @property vs. Descriptors

Feature	**@property**	**Descriptors**
Ease of Use	Easy to implement and read.	Requires understanding of __get__, __set__, __delete__.
Reusability	Tied to a single class.	Can be reused across multiple classes.
Granularity	Inline getter, setter, and deleter.	Separate methods, offering more control.
Stateful Logic	No built-in state management.	Can store state via additional attributes.

When to Use
Use @property when:
- Attribute logic is class-specific.
- You want a simple, inline syntax.

Use descriptors when:
- Logic needs to be reused across classes.
- Stateful or complex attribute management is required.

property() - @property

The @property decorator simplifies attribute management by wrapping methods to behave like attributes. This avoids breaking the class interface when additional logic is needed for attribute access. Using propert() as a function rather than in it's decorator form is an option, but not seen as often.

Advantages: Inline getter, setter, and deleter functionality, simple syntax, transparent usage for clients of the class.

Read-Only Attributes: Attributes that should be accessible but not modifiable

```
class User:
    def __init__(self, username):
        self._username = username

    @property
    def username(self):
        return self._username

user = User("JohnDoe")
print(user.username)      # Outputs: JohnDoe
```

Dynamic Computations: Attributes that compute values dynamically

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * (self.radius ** 2)

circle = Circle(5)
print(circle.area)              # Outputs: 78.54
```

Getter Methods- Used to compute or access protected attributes

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def area(self):             #  Compute area dynamically
        return self._width * self._height

# Usage
r = Rectangle(3, 4)
print(r.area)                   #  Outputs: 12
```

Controlled Updates (Setters): Validate or preprocess values during updates

```
class Square:
    def __init__(self, side):
        self._side = side

    @property
    def side(self):
        return self._side

    @side.setter
    def side(self, value):
        if value < 0:
            raise ValueError("Side must be non-negative")
        self._side = value

square = Square(4)
square.side = 6
print(square.side)              # Outputs: 6
```

Deleter Methods- Enable controlled deletion of attributes

```
class Counter:
    def __init__(self):
        self._count = 0

    @property
    def count(self):
        return self._count

    @count.deleter
    def count(self):
        print("Deleting count")
        del self._count

# Usage
c = Counter()
del c.count                    # Outputs: Deleting count
```

Descriptors

Descriptors provide a lower-level mechanism for controlling attribute behavior through special methods: __get__, __set__, and __delete__

The labels "data descriptors" and "non-data descriptors" don't seem logical . They refer to control and priority over attributes rather than whether they "handle data." Data Descriptors implement both __set__ and/or __delete__ along with __get__, so they define how data is managed more than a "non-data descriptor" which simply retrieves data.
One topic which is beyond the basics below is that you can make custom descriptors that use your own class.

Data Descriptors (standard) - Implement both __get__ and __set__:

```
class DataDescriptor:
    def __get__(self, instance, owner):
        return instance._value

    def __set__(self, instance, value):
        instance._value = value

class MyClass:
    attribute = DataDescriptor()

# Usage
obj = MyClass()
obj._value = 10
print(obj.attribute)           # Outputs: 10
```

Non-Data Descriptor (standard) - Implement only __get__ (read-only behavior):

```
class NonDataDescriptor:
    def __get__(self, instance, owner):
        return "Read-only attribute"

class MyClass:
    attribute = NonDataDescriptor()

# Usage
obj = MyClass()
print(obj.attribute)           # Outputs: Read-only attribute
```

A "DeletionOnlyDescriptor" - (self-explanitory)

```
class DeletionOnlyDescriptor:
    def __delete__(self, instance):
        print(f"Deleting attribute for {instance}")

class MyClass:
    attribute = DeletionOnlyDescriptor()

# Usage
obj = MyClass()
del obj.attribute              # Outputs: Deleting attribute for <__main__.MyClass object at 0x...>
```

An All-in-One Data Descriptor - demonstrates the full lifecycle of managing an attribute, including deletions

```
class AllDataDescriptor:
    def __init__(self, default=None):
        self.default = default                  # Default value if no data is set for an instance
        self.data = {}                          # A dictionary to store instance-specific data

    def __get__(self, instance, owner):
        if instance is None:
            return self  # Accessed via the class; returns the descriptor itself
        return self.data.get(instance, self.default)        # Instance-specific or default

    def __set__(self, instance, value):
        self.data[instance] = value                         # Sets instance-specific data

    def __delete__(self, instance):
        if instance in self.data:
            print(f"Deleting value for {instance}")
            del self.data[instance]                         # Deletes instance-specific data
        else:
            raise AttributeError("Attribute does not exist or has not been set")

class MyClass:
    attribute = AllDataDescriptor(default="Default Value")

# Usage
obj = MyClass()
print(obj.attribute)  # Outputs: Default Value (uses default from __init__)
obj.attribute = "Custom Value"
print(obj.attribute)  # Outputs: Custom Value (stored in self.data)
del obj.attribute     # Deletes the instance-specific value
print(obj.attribute)  # Outputs: Default Value (falls back to default)
```

__get__: Retrieves the stored value for the instance or returns a default value if none is set.
__set__: Stores the provided value in a dictionary, keyed by the instance.
__delete__: Deletes stored value for instance and handles attempts to delete non-existent attributes by raising an exception.

Advantages of Descriptors

- Reusable attribute logic across multiple classes.
- Fine-grained control over attribute behavior.
- Capability to manage state associated with attributes.
- Useful for advanced scenarios like metaprogramming or custom validation.

Combining @property and Descriptors

These two approaches can work together to provide both ease of use and reusability.
This hybrid approach ensures flexibility, allowing simple syntax while leveraging descriptor-like logic.

Example: Temperature Conversion

```
class Temperature:
    def __init__(self):
        self._celsius = 0

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        self._celsius = value

    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

temp = Temperature()
temp.celsius = 25
print(temp.fahrenheit)  # Outputs: 77.0
```

Using the functools.wraps @wraps decorator

When you decorate a function, the decorator replaces the original function with a wrapper, which by default loses important metadata like __name__, __doc__, and annotations. This can hinder debugging, introspection, and docstrings.
The functools.wraps decorator ensures the wrapper function preserves the original function's metadata, improving transparency and compatibility with tools.

```
from functools import wraps
def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with arguments {args}")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def say_hello(name: str) -> str:       #  Returns a greeting for the given name."""
    return f"Hello, {name}!"

# Function retains original metadata
print(say_hello.__name__)          #  Output: say_hello
print(say_hello.__doc__)           #  Output: Returns a greeting for the given name.
```

Without @wraps, say_hello.__name__ would be wrapper, and __doc__ would be lost.
@wraps is a shortcut for functools.update_wrapper, which manually copies attributes like __name__, __doc__, and __annotations__ from the original function to the wrapper.

```
from functools import update_wrapper
def my_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    update_wrapper(wrapper, func)
    return wrapper
```

Built-in Functions for Classes

isinstance():
Checks if an object is an instance or a tuple of classes.

```
print(isinstance(5, int))  # True
print(isinstance("text", (int, str)))  # True
```

issubclass():
Checks if a class is a subclass of another class.

```
class Animal: pass
class Dog(Animal): pass
print(issubclass(Dog, Animal))  # True
```

{has, get, set, del}attr():
These functions work with object attributes:
hasattr(): Checks if an object has an attribute.
getattr(): Retrieves the value of attribute (optional default).
setattr(): Sets an attribute on an object.
delattr(): Deletes an attribute from an object.

```
class Example:
    value = 42

obj = Example()
print(hasattr(obj, "value"))  # True
print(getattr(obj, "value", "default"))  # 42
setattr(obj, "new_attr", 100)
print(obj.new_attr)  # 100
delattr(obj, "value")
```

dir():
Returns a list of attributes (methods, properties) of an object.

```
class Example:
    value = 42
print(dir(Example))
# ['__class__', '__delattr__', ..., 'value']
```

Built-in Functions for Functions

callable():
Checks if an object is callable (e.g., a function, method, or class with __call__ defined).

```
print(callable(len))  # True
print(callable(42))  # False
```

id():
Returns the memory address of an object. Useful for understanding object identity.

```
x = [1, 2, 3]
print(id(x))  # Memory address of x
```

type():
Returns the type of an object or creates a new type if used with additional arguments.

```
print(type(42))  # <class 'int'>
```

vars():
Returns the __dict__ attribute of an object, if it exists (a dictionary of attributes).

```
class Example:
    value = 42
obj = Example()
print(vars(obj))  # {}
```

Built-in Functions for Variable Management

globals():
Returns a dictionary of the global namespace.

```
x = 10
print(globals())  # Dictionary containing 'x'
```

locals():
Returns a dictionary of the local namespace.

```
def func():
    local_var = 5
    print(locals())  # {'local_var': 5}
func()
```

eval():
Evaluates a string expression within the current scope.

```
x = 5
print(eval("x + 3"))  # 8
```

exec():
Executes dynamically created Python code.

```
code = "y = 10\nprint(y)"
exec(code)  # Prints 10
```

compile():
Compiles a string into a code object for later execution.

```
code = compile("z = 15", "<string>", "exec")
exec(code)
print(z)  # 15
```

hash():
Returns the hash value of an object.

```
print(hash("example"))  # Hash value of the string
```

__Dunder__ Methods

For Classes

__init__(self, ...):
Initializes a new instance of a class.

```
class Example:
    def __init__(self, value):
 self.value = value
obj = Example(10)
print(obj.value)  # 10
```

__new__(cls, ...):
Allocates and returns a new instance.

```
class Example:
    def __new__(cls):
 print("Creating instance")
 return super().__new__(cls)
obj = Example()  # Prints "Creating instance"
```

__repr__(self):
Returns a string representation of object (used by repr()).

```
class Example:
    def __repr__(self):
 return "Example()"
print(repr(Example()))  # Example()
```

__str__(self):
Returns readable representation (used by str() , print()).

```
class Example:
    def __str__(self):
 return "This is an example."
print(Example())  # This is an example.
```

__call__(self, *args, **kwargs):
Makes an instance callable like a function.

```
class Example:
    def __call__(self, x):
 return x * 2
obj = Example()
print(obj(5))  # 10
```

For Functions

__call__(self, *args, **kwargs):
Makes a custom object callable.

```
class CallableObject:
    def __call__(self, x):
 return x + 1
obj = CallableObject()
print(obj(5))  # 6
```

For Attribute Management

__getattr__(self, name):
Called when accessing an attribute that doesn't exist.

```
class Example:
    def __getattr__(self, name):
 return f"{name} is not found!"
obj = Example()
print(obj.missing_attr)  # missing_attr is not found!
```

__setattr__(self, name, value):
Customizes setting an attribute.

```
class Example:
    def __setattr__(self, name, value):
 print(f"Setting {name} to {value}")
 super().__setattr__(name, value)
obj = Example()
obj.attr = 10  # Setting attr to 10
```

__delattr__(self, name):
Customizes deleting an attribute.

```
class Example:
    def __delattr__(self, name):
 print(f"Deleting {name}")
 super().__delattr__(name)
obj = Example()
obj.attr = 10
del obj.attr  # Deleting attr
```

For General Objects

__repr__(self):
Provides an unambiguous string representation of the object. Often required for debugging.

__str__(self):
Provides a human-readable representation of the object.

__len__(self):
Defines behavior for len(obj).

```
class Example:
    def __len__(self):
 return 5
print(len(Example()))  # 5
```

__getitem__(self, key):
Enables indexing and slicing for objects.

```
class Example:
    def __getitem__(self, key):
 return key * 2
obj = Example()
print(obj[3])  # 6
```

__setitem__(self, key, value):
Enables setting values via indexing.

__delitem__(self, key):
Enables deleting items via indexing.

__enter__(self) and __exit__(self):
Defines behavior for context management (e.g., with statements).

```
class MyContext:
    def __enter__(self):
        print("Entering the context")
        return self  # Optional, can return another object

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")

        # Using the context manager
with MyContext() as ctx:
    print("Inside the context")
        # Output: Entering the context
        # Inside the context
        # Exiting the context
```

Dataclasses module

Simplifies creation of data containers

Basic Data Class

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

p = Point(10, 20)
print(p.x, p.y)  # Outputs: 10 20
```

Default factory

```
from dataclasses import dataclass, field

@dataclass
class Inventory:
    items: list[str] = field(default_factory=list)

inv = Inventory()
inv.items.append("Sword")
print(inv.items)  # Outputs: ['Sword']
```

Comparison methods

```
from dataclasses import dataclass

@dataclass(order=True)
class Task:
    priority: int
    name: str

t1 = Task(1, "Write Code")
t2 = Task(2, "Test Code")
print(t1 < t2)  # Outputs: True
```

Attrs module

Enhances data class functionality with validation and more.

Basic usage:

```
from attrs import define, validators

@define
class Person:
    name: str
    age: int = validators.instance_of(int)
```

```
person = Person("Alice", 30)
```

Default values with factories:
```
from attrs import define, Factory

@define
class Inventory:
    items: list = Factory(list)

inv = Inventory()
inv.items.append("Potion")
print(inv.items)  # Outputs: ['Potion']
```

Custom conversion:
```
from attrs import define, converters

@define
class Product:
    price: float = converters.optional(float)  # Ensures input is converted to float

product = Product(price="19.99")
print(product.price)  # Outputs: 19.99
```

Toolz module

Functional programming utilities.

Functional composition:
```
from toolz import compose

def double(x):
    return x * 2

def increment(x):
    return x + 1

double_then_increment = compose(increment, double)
print(double_then_increment(3))  # Outputs: 7
```

Groupby:
```
from toolz import groupby

data = [{"type": "fruit", "name": "apple"}, {"type": "fruit", "name": "banana"}, {"type": "vegetable", "name": "carrot"}]
grouped = groupby("type", data)
print(grouped)
# Outputs: {'fruit': [{'type': 'fruit', 'name': 'apple'}, {'type': 'fruit', 'name': 'banana'}], 'vegetable': [{'type': 'vegetable', 'name': 'carrot'}]}
```

Sliding window:
```
from toolz import sliding_window

data = [1, 2, 3, 4, 5]
print(list(sliding_window(2, data)))
# Outputs: [(1, 2), (2, 3), (3, 4), (4, 5)]
```

Partition:
```
from toolz import partition

data = [1, 2, 3, 4, 5, 6]
print(list(partition(2, data)))
# Outputs: [(1, 2), (3, 4), (5, 6)]
```

functools

Module specializes in higher-order functions and optimizations, enabling more efficient code.

partial: **p**re-defines some arguments of a function, returning a new callable with fixed values.

```
from functools import partial
def multiply(x, y):
    return x * y

double = partial(multiply, 2)
print(double(5))  # Outputs: 10
```

partialmethod: Similar to partial, but specifically for methods in classes. It allows fixing certain arguments or keywords of methods in a class.

```
from functools import partialmethod

class Greeter:
    def greet(self, greeting, name):
        return f"{greeting}, {name}!"

    say_hello = partialmethod(greet, "Hello")

g = Greeter()
print(g.say_hello("Alice"))  # Hello, Alice!
```

singledispatch

Creates generic functions that can be specialized based on argument types.

```
from functools import singledispatch

@singledispatch
def process(data):
    raise NotImplementedError("Unsupported type")

@process.register(str)
def _(data):
    return data.upper()

@process.register(int)
def _(data):
    return data * 10

print(process("hello"))  # Outputs: HELLO
print(process(5)) # Outputs: 50
```

cmp_to_key: Converts an old-style comparison function (cmp) into a key function for sorting.

```
from functools import cmp_to_key

def compare(a, b):
    return (a > b) - (a < b)

numbers = [3, 1, 4, 1, 5, 9]
sorted_numbers = sorted(numbers,
key=cmp_to_key(compare))
print(sorted_numbers)  # [1, 1, 3, 4, 5, 9]
```

lru_cache: Implements memoization to cache results of expensive or frequently called functions.

```
from functools import lru_cache
@lru_cache(maxsize=128)
def factorial(n):
        return 1 if n == 0 else n * factorial(n - 1)
print(factorial(5))  # Outputs: 120
```

cached_property: Converts a method into a property whose value is computed once and then cached for the lifetime of the instance.

```
from functools import cached_property

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @cached_property
    def area(self):
        print("Calculating area...")
        return 3.14159 * self.radius ** 2

c = Circle(5)
print(c.area)  # Calculating area... \n 78.53975
print(c.area)  # 78.53975 (cached, no recalculation)
```

reduce: **s**uccessively applies a function to the elements of an iterable, reducing it to a single value.

```
from functools import reduce

numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Outputs: 24
```

wraps: preserves the metadata of the original function when wrapped in a decorator.

```
from functools import wraps

def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
 print(f"Calling {func.__name__}")
 return func(*args, **kwargs)
    return wrapper

@decorator
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))  # Outputs: Calling greet\nHello, Alice!
```

Pydantic

pydantic combines type annotations with data validation, ensuring correctness in structured data.

Data models:

```
from pydantic import BaseModel

class Product(BaseModel):
    id: int
    name: str
    price: float

product = Product(id=1, name="Laptop", price=999.99)
print(product.dict())  # Outputs: {'id': 1, 'name': 'Laptop', 'price': 999.99}
```

Validation:

```
from pydantic import BaseModel, ValidationError
class User(BaseModel):
    username: str
    age: int
try:
    user = User(username="Alice", age="twenty")
except ValidationError as e:
    print(e)
    # Outputs: age\n  value is not a valid integer (type=type_error.integer)
```

Custom validation:

```
from pydantic import BaseModel, validator

class User(BaseModel):
    username: str
    age: int

    @validator("age")
    def check_age(cls, value):
 if value < 18:
     raise ValueError("Age must be 18 or above.")
 return value

user = User(username="Bob", age=20)  # Valid
# user = User(username="Bob", age=16)  # Raises validation error
```

Complex nested models:

```
from pydantic import BaseModel
user = User(username="Alice", email="alice@example.com", address={"street": "123 Main St", "city": "Townsville", "zipcode":
"12345"})
class Address(BaseModel):
    street: str
    city: str
    zipcode: str

class User(BaseModel):
    username: str
    email: str
    address: Address

print(user.dict())  # Outputs nested dictionary with validated data
```

Strict types:

```
from pydantic import BaseModel, StrictInt

class Config(BaseModel):
    setting: StrictInt  # Only allows integers; floats are rejected

# config = Config(setting=3.14)  # Raises validation error
config = Config(setting=42)
```