Python Fundamentals: Built-in Functions

Unlocking Python's Core Power Without External Libraries

Bîcu Andrei Ovidiu

What Are Built-in Functions?

Python's built-in functions are globally available utilities provided by the Python interpreter without needing imports.

```
# All of these are built-in functions
print(len("hello"))          # 5
print(any([0, 0, 1]))         # True
print(type([]))              # <class 'list'>
```

Key Facts:

- You can use them without importing any modules
- Found in Python's global namespace (builtins)
- Enhance productivity, readability, and performance

Example of Built-ins:

- len(), map(), eval(), zip(), type(), any()

Built-in Functions in Python

abs()	classmethod()	filter()	id()	max()	property()	str()
all()	compile()	float()	input()	memoryview()	range()	sum()
any()	complex()	format()	int()	min()	repr()	super()
ascii()	delattr()	frozenset()	isinstance()	next()	reversed()	tuple()
bin()	dict()	getattr()	issubclass()	object()	round()	type()
bool()	dir()	globals()	iter()	oct()	set()	vars()
bytearray()	divmod()	hasattr()	len()	open()	setattr()	zip()
bytes()	enumerate()	hash()	list()	ord()	slice()	__import__()
callable()	eval()	help()	locals()	pow()	sorted()	
chr()	exce()	hex()	map()	print()	staticmethod()	

Categories of Built-in Functions

Category	***Example Functions***	***Purpose***
Functional Tools	map(), filter(), zip(), any()	Transform and evaluate sequences
Evaluation / Introspection	eval(), globals(), dir()	Inspect or execute code dynamically
Type Checking	type(), isinstance(), callable()	Object analysis
Conversion	int(), str(), list()	Cast between types
I/O	print(), input()	Handle user interaction
Advanced Utilities	slice(), memoryview(), help()	Power tools

Functional Tools

map()	Apply a function to every item in an iterable
filter()	Keep only items that match a condition

```python
nums = [1, 2, 3, 4, 5]

# Double each number
doubled = list(map(lambda x: x* 2, nums)) # [2, 4, 6, 8, 10]

# Keep only even numbers
evens = list(filter(lambda x: x % 2 == 0, nums)) # [2, 4]
```

Iteration Tools

zip()	Combine multiple iterables element-wise
enumerate()	Attach an index to each item in an iterable

```
'''zip()'''

names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]

combined = list(zip(names, scores))
print(combined) # [('Alice', 85), ('Bob', 92), ('Charlie', 78)]

# Unzippint with zip(*zipped)
zipped = list(zip(['a', 'b'], [1, 2]))
names, values = zip(*zipped)

print(names) # ('a', 'b')
print(values) # (1, 2)

# common gotcha -- Behavior on Uneven Lenghts (zip() truncates)
names = ["Alice", "Bob"]
scores = [85, 92, 78]

print(list(zip(names, scores)))
# [('Alice', 85), ('Bob', 92)]  ← Notice 78 is dropped!

'''enumerate()'''
for idx, name in enumerate(names, start = 1):
    print(f"{idx}. {name}")
# 1. Alice
# 2. Bob
# 3. Charlie
```

Caution: on uneven lengths you can use itertools.zip_longest() instead of zip() – It solves this by continuing until the **longest** iterable is exhausted. Missing values are filled with a default (like None)

Logical Built-ins

Function	Returns True if…
all()	**All** elements in the iterable are truthy
any()	**At least one** element in the iterable is truthy

```
nums = [1, 2, 3, 0]

print(all(nums)) # False - 0 is Falsy
print(any(nums)) # True - at least one value is truthy

conditions = [x > 0 for x in nums]
print(all(conditions)) # False because of 0
```

These functions are clean, fast replacements for loops like:

```
for x in items:
    if not x: return False
```

Dynamic Execution

Caution: These are dangerous if used with untrusted input!!!

eval()	Evaluates a Python expression string
exec()	Executes a block of code from a string
compile()	Compiles code to be used with eval()/exec()

```
'''eval() vs exec()'''
x = 10

# eval() evaluates a string as a Python *expression*
# Safe here because we're controlling the input
print(eval("x + 5"))     # Output: 15

# exec() executes a string as a block of Python *code*
# It can include assignments, loops, function defs, etc.
exec("y = x * 2")        # Defines y in the current scope
print(y)                 # Output: 20

'''compile()'''
# compile() can convert source code into a code object
# Use mode='eval' for expressions (can also use 'exec' or 'single')
code = "x + 3"

compiled = compile(code, "<string>", "eval")  # Filename is optional label
print(eval(compiled))    # Output: 13 – safely evaluates the precompiled code
```

Introspection Tools

dir()	Lists attributes/methods of an object
vars()	Returns __dict__ of an object (if present)
locals()	Returns current local symbol table (dict)
globals()	Returns current global symbol table (dict)

```python
# Define a simple class
class Person:
    def __init__(self, name):
        self.name = name

# Create an instance
p = Person("Alice")

# dir() lists attributes and methods of an object
# Includes inherited and dunder methods (like __init__, __str__, etc.)
print(dir(p))

# vars() returns the instance's __dict__ (its attributes as a dict)
print(vars(p))  # Output: {'name': 'Alice'}

# locals() returns a dictionary of the local symbol table
# This will include all local variables in the current scope
print(locals())  # Outputs a large dict with everything in local scope

# globals() returns a dictionary of the global namespace
# Useful for accessing or checking variables defined at the top level
print('Person' in globals())  # True – 'Person' class is defined globally
```

Type Checking

type(obj)	Returns the type of an object
isinstance(obj, type)	Checks if object is an instance of a given type or tuple of types

```
x = 42
y = "hello"

# type() returns the exact type of the object
print(type(x))         # Output: <class 'int'>
print(type(y))         # Output: <class 'str'>

# isinstance() checks if object matches a given type
print(isinstance(x, int))          # True
print(isinstance(y, (int, float))) # False – not a number
```

Dynamically Working with Objects

hasattr(obj, name)	Returns True if the object has an attribute with the given name
getattr(obj, name[, default])	Retrieves the value of the named attribute (optionally with a fallback)
setattr(obj, name, value)	Sets the named attribute on the object to the given value
delattr(obj, name)	Deletes the named attribute from the object

```python
class Person:
    def __init__(self):
        self.name = "Alice"

p = Person()

# hasattr checks if 'name' exists
print(hasattr(p, "name"))      # True

# getattr retrieves 'name'
print(getattr(p, "name"))      # Alice

# getattr with fallback for non-existent attribute
print(getattr(p, "age", 30))   # 30

# setattr creates/updates 'age'
setattr(p, "age", 25)
print(p.age) # type: ignore     # 25

# delattr deletes 'name'
delattr(p, "name")
print(hasattr(p, "name"))      # False
```

Type Conversion Built-ins

int(x)	Converts x to an integer
float(x)	Converts x to a floating-point number
str(x)	Converts x to a string
bool(x)	Converts x to a Boolean
complex(x)	Converts x to a complex number

```
# int() converts string to integer
print(int("10"))        # 10

# float() from string
print(float("3.14"))    # 3.14

# str() turns numbers into strings
print(str(25))          # "25"

# bool() converts to boolean - empty list is False
print(bool([]))         # False

# complex() with integer input
print(complex(4))       # (4+0j)

# Watch out: float to int truncates, doesn't round!
print(int(3.99))        # 3
```

Collection Constructors

Function	Purpose	Literal Equivalent
list()	Creates a list	[]
tuple()	Creates a tuple	()
set()	Creates a set (unordered, unique values)	{} (set is ambiguous literal)
dict()	Creates a dictionary	{}
frozenset()	Creates an immutable version of a set	*No literal syntax*

```
# Create a list from a string
print(list("abc"))          # ['a', 'b', 'c']

# Convert list to tuple
print(tuple([1, 2, 3]))     # (1, 2, 3)

# Create set from a list (removes duplicates)
print(set([1, 2, 2, 3]))    # {1, 2, 3}

# Create dictionary from pairs
print(dict([("a", 1), ("b", 2)]))  # {'a': 1, 'b': 2}

# Create a frozenset (immutable set)
f = frozenset([1, 2, 2, 3])
print(f)                    # frozenset({1, 2, 3})

# Caution: {} is an empty dict, not an empty set!
print(type({}))             # <class 'dict'>
```

String Representation and Formatting

Function	Purpose	Example
repr(x)	Returns an official, unambiguous string representation	repr("hi") → "'hi'"
ascii(x)	Like **repr()** but escapes non-ASCII characters	ascii("ñ") → "'\\u00f1'"
format(x)	Custom string formatting via format specifiers	format(3.14159, ".2f") → "3.14"

```
# repr() gives detailed string (good for debugging)
text = "hello\nworld"
print(repr(text))            # 'hello\nworld'

# ascii() escapes non-ASCII safely
emoji = "café 🧉"
print(ascii(emoji))          # 'caf\\xe9 \\u2615'

# format() with float precision
pi = 3.14159265
print(format(pi, ".2f"))     # '3.14'

# format() for padding and alignment
n = 42
print(format(n, "04"))       # '0042'
print(format(n, "<6"))       # '42    '

# Also works with named placeholders
template = "Name: {name}, Age: {age}"
print(template.format(name="Ada", age=30))  # 'Name: Ada, Age: 30'
```

Bîcu Andrei Ovidiu

User Interaction

Function	Purpose	Notes
print()	Sends output to the standard stream	Supports multiple values and sep, end
input()	Gets a string input from the user	Always returns a string!

```python
# Basic usage
print("Hello, world!")           # Hello, world!

# Multiple arguments and custom separator
print("A", "B", "C", sep="-")    # A-B-C

# Suppressing the newline
print("Loading...", end="")
print("Done")                    # Output: Loading...Done

# Asking for input
name = input("What is your name? ")
print("Nice to meet you,", name)

# Always returns a string, even if you type a number!
age = input("Enter your age: ")
print("Age plus one:", int(age) + 1)
```

Advanced Data Access

Function	Purpose	Common Use Case
reversed(seq)	Returns a reverse iterator over a sequence	For loops, reverse traversal
slice(start, stop, step)	Creates a reusable slicing object	Dynamic or reusable slicing
memoryview(obj)	Views the memory of bytes-like objects	Efficient I/O or binary data access

```python
# reversed(): iterates from end to start
for ch in reversed("Python"):
    print(ch, end=" ")            # n o h t y P

# slice(): reusable slicing logic
data = list(range(10))            # [0, 1, 2, ..., 9]
s = slice(2, 8, 2)
print(data[s])                    # [2, 4, 6]

# memoryview(): view byte data without copying
b = bytearray(b"abcde")
m = memoryview(b)
print(m[1])                       # 98 (ASCII for 'b')

# Modify through memoryview
m[1] = 122
print(b)                          # bytearray(b'azcde')
```

Bîcu Andrei Ovidiu

Dynamic Import

Function	Purpose	Notes
help(obj)	Shows the help text/documentation	Works on functions, modules, classes
__import__(name)	Imports a module dynamically	Rarely used directly; behind import

```
# help(): get docs for any object
help(str)                       # Shows all string methods and docstrings
help(len)                       # Shows the function's purpose and usage

# __import__(): dynamically import modules by name
mod_name = "math"
math_mod = __import__(mod_name)
print(math_mod.sqrt(16))        # Output: 4.0

# Dynamic import is useful in plugins, CLI tools, etc.
```

Common Pitfalls

- Never use eval() with untrusted input. Use ast.literal_eval() for safer evaluation of literals.
- all() **returns True for empty input (vacuous truth)**
- any() **returns False for empty input.**
- zip() silently drops unmatched elements.
    - Use itertools.zip_longest() if you want to keep all values (not a built-in)

```
'''eval() Misuse - Security Warning'''
user_input = "2 + 2"
print(eval(user_input))         # Works, but dangerous!

# If user_input = "__import__('os').system('rm -rf /')"
# This could run arbitrary code — big security risk!

'''all() returns True'''
print(all([]))      # True
print(any([]))      # False


'''zip() Truncates to the shortest iterable'''
a = [1, 2, 3]
b = ['a', 'b']
print(list(zip(a, b)))      # [(1, 'a'), (2, 'b')]
```

Best Practices

- **Use Built-ins Before External Libraries**

  Built-ins are:

  - Fast (written in C under the hood)
  - Tested and maintained by the Python core team
  - Require no extra dependencies.
- **Prefer Readable & Idiomatic Usage**
  - Built-ins often express your intent better and concisely.
- **Avoid Overusing Obscure Built-ins**
  - Use only if dynamic behavior is required.
- **Avoid Shadowing Built-ins**
  - Renam such variables (my_list, lst, etc.)

```
'''Use built-ins before external libraries'''
# Prefer this:
squared = list(map(lambda x: x**2, range(5)))

# Over external libraries unless truly necessary

'''Prefer readable & idiomatic usage'''
# Clear and Pythonic
total = sum(numbers) # type: ignore

# Less readable
total = 0
for num in numbers:  # type: ignore
    total += num

'''Avoid overusing obscure built-ins'''
# __import__ is powerful, but rarely needed directly
mod = __import__('math')
print(mod.sqrt(25))

# prefer import math

'''Avoid shadowing built-ins'''
#  Bad practice: naming a variable after a built-in
list = [1, 2, 3]     # Now you can't use the list() function!
```

Cheat Sheet

Name	Purpose	Safe to Use?	Common Use Case
hasattr()	Check if attribute exists	✅ Yes	Dynamic attribute checks
getattr()	Get attribute dynamically	✅ Yes	Flexible object access
setattr()	Set attribute dynamically	✅ Yes	Dynamic object modification
delattr()	Delete attribute	✅ Yes	Cleanup or metaprogramming
int(), float()	Convert to number	✅ Yes	User input parsing, math
str(), bool()	Convert to string or boolean	✅ Yes	Display, condition checks
complex()	Complex numbers	✅ Yes	Scientific computing
list(), tuple()	Create sequences	✅ Yes	Data conversion and processing
set(), frozenset()	Unordered unique collections	✅ Yes	Fast membership tests
dict()	Key-value mappings	✅ Yes	Data structures, configs
repr(), ascii()	Represent objects as strings	✅ Yes	Debugging, safe display

Name	Purpose	Safe to Use?	Common Use Case
format()	Custom string formatting	✅ Yes	Output formatting
print()	Output to console	✅ Yes	User communication
input()	Read user input	✅ Yes	Interactive programs
reversed()	Iterate backwards	✅ Yes	Reverse iteration
slice()	Create slice objects	✅ Yes	Dynamic slicing
memoryview()	Memory-efficient byte access	✅ Yes	High-performance binary data
help()	Show documentation	✅ Yes	Learning and debugging
__import__()	Dynamic module import	⚠️ Use sparingly	Plugins, dynamic imports
eval()	Evaluate code strings	❌ Avoid	Security risk, avoid if possible
all(), any()	Check truthiness in iterables	✅ Yes	Condition checking
zip()	Pair items from iterables	✅ Yes	Parallel iteration

Bîcu Andrei Ovidiu

Glossary

Term	Definition
Lazy Evaluation	Delaying computation until the result is needed, improving performance and resource use
Introspection	The ability of a program to examine the type or properties of an object at runtime
Coercion	Automatic or explicit conversion of one data type to another
Vacuous Truth	A statement that is considered true because there are no counterexamples (e.g., all([]))
Metaprogramming	Writing code that manipulates code or program structure dynamically
Immutable	An object whose state cannot be modified after creation (e.g., frozenset)
Iterable	An object capable of returning its members one at a time, allowing it to be looped over
Iterator	An object representing a stream of data, returned by calling iter() on an iterable
Callable	An object that can be called like a function (e.g., functions, classes with __call__ method)
Docstring	A string literal used to document modules, classes, or functions
Dynamic Import	Importing modules during runtime rather than at compile time
Shadowing	Defining a variable with the same name as a built-in, hiding the original function or object

Bîcu Andrei Ovidiu

References & Further Reading

Recommended Books & Documentation

- **Official Python Docs -** https://docs.python.org/3/library/functions.html
- *Fluent Python* - Luciano Ramalho (O'Reilly) - Excellent deep dive on Python idioms
- *Effective Python* **-** Brett Slatkin - Tips and best practises, including built-ins

Online Tutorials & Videos

- Corey Shafer's Python Tutorial Series - https://www.youtube.com/@coreyms/playlists
- Real Python: Python Built-ins Explained - https://realpython.com/python-built-in-functions

Practice & Interactive Learning

- Practice problems with mentor feedback - **https://exercism.org**
- LeetCode - Coding challenges to apply built-ins