A Comprehensive Guide to Python's Built-in Functions

In Python, "built-in" means a function is available globally—you can use it anywhere in your code without needing to `import` any module. These functions form the fundamental building blocks for performing common tasks.

This guide categorizes the most important built-in functions to help you understand their purpose and when to use them.

Table of Contents

---

1. Type Conversion

These functions are used to convert data from one type to another.

Function	What it does
`int(x, base=10)`	Converts a string or number to an integer.
`float(x)`	Converts a string or number to a floating-point number.
`str(object)`	Converts an object to its string representation.
`list(iterable)`	Converts an iterable (like a tuple or set) to a list.
`tuple(iterable)`	Converts an iterable to a tuple.
`set(iterable)`	Converts an iterable to a set (removing duplicates).
`dict(...)`	Creates a dictionary.
`bool(x)`	Converts a value to a boolean (`True` or `False`).

Examples

```python
# From string to number
num_str = "101"
x = int(num_str)        # x is 101 (integer)
y = float(num_str)      # y is 101.0 (float)

# Convert a binary string to int
z = int("101", base=2)  # z is 5

# From number to string
```

```python
pi_str = str(3.14)      # pi_str is "3.14"

# From one collection to another
my_tuple = (1, 2, 2, 3)
my_list = list(my_tuple)  # [1, 2, 2, 3]
my_set = set(my_tuple)    # {1, 2, 3} (duplicates removed)

# bool() evaluates "truthiness"
print(bool(0))          # False
print(bool([]))         # False (empty collections are False)
print(bool("hello"))    # True
```

Why & When to Use: This is one of the most common tasks. Use `int()` and `float()` to process user input from `input()`. Use `list()`, `set()`, and `tuple()` to change the mutability or properties of a collection (e.g., use `set()` to quickly get unique items from a list).

---

2. Working with Iterables

These functions operate on sequences like lists, tuples, and strings.

Function	What it does
`len(s)`	Returns the number of items in a container.
`sum(iterable, start=0)`	Sums the items of an iterable.
`min(iterable)`	Returns the smallest item in an iterable.
`max(iterable)`	Returns the largest item in an iterable.
`sorted(iterable, key=None, reverse=False)`	Returns a **new** sorted list from the items in an iterable.
`reversed(seq)`	Returns a reverse **iterator**.
`enumerate(iterable, start=0)`	Returns an iterator of pairs `(index, item)`.
`zip(*iterables)`	Returns an iterator that aggregates elements from each of the iterables.
`all(iterable)`	Returns `True` if all elements of the iterable are truthy.
`any(iterable)`	Returns `True` if any element of the iterable is truthy.
`map(function, iterable)`	Applies a function to every item of an iterable and returns an iterator of the results.
`filter(function, iterable)`	Constructs an iterator from elements of an iterable for which a function returns `True`.

Examples

```python
nums = [3, 1, 4, 1, 5, 9, 2]
```

```python
# Basic stats
print(len(nums))   # 7
print(sum(nums))   # 25
print(min(nums))   # 1
print(max(nums))   # 9

# Sorting
# sorted() returns a new list, does not modify the original
sorted_nums = sorted(nums) # [1, 1, 2, 3, 4, 5, 9]
# Note: This is different from `nums.sort()`, which sorts the list in-place.

# Looping helpers
names = ['Alice', 'Bob', 'Charlie']
for i, name in enumerate(names):
    print(f"{i}: {name}") # 0: Alice, 1: Bob, 2: Charlie

scores = [95, 88, 76]
for name, score in zip(names, scores):
    print(f"{name} scored {score}") # Alice scored 95, ...

# map and filter (often replaced by list comprehensions)
# Using map
str_nums = ['1', '2', '3']
int_nums = list(map(int, str_nums)) # [1, 2, 3]

# Using filter
def is_even(n): return n % 2 == 0
evens = list(filter(is_even, nums)) # [4, 2]
# The list comprehension equivalent is often preferred: [n for n in nums if n % 2 ==
0]

# all() and any()
checks = [True, True, False]
print(all(checks)) # False
print(any(checks)) # True
```

Why & When to Use: This category is the bread and butter of data processing. Use `len()` for checking size, `sum()` / `min()` / `max()` for quick stats. Use `enumerate()` and `zip()` to write cleaner, more Pythonic `for` loops. Use `sorted()` whenever you need a sorted copy of a collection.

---

3. Input & Output

Function	What it does
`print(*objects, sep=' ', end='\n', file=sys.stdout)`	Prints objects to the text stream (e.g., the console).
`input(prompt)`	Reads a line from input, converts it to a string, and returns it.

Examples

```python
# print()
print("Hello", "World", sep="---") # Hello---World
print("First line", end=" ")
print("Second line") # Prints on the same line: First line Second line

# input()
# Note: input() ALWAYS returns a string!
name = input("What is your name? ")
age_str = input("What is your age? ")
try:
    age = int(age_str)
    print(f"Hello, {name}! You will be {age + 1} next year.")
except ValueError:
    print("Invalid age entered.")
```

---

4. Mathematical Functions

Function	What it does
`abs(x)`	Returns the absolute value of a number.
`round(number, ndigits=None)`	Rounds a number to a given precision in decimal digits.
`pow(base, exp)`	Returns `base` to the power of `exp`. Same as `base ** exp`.
`divmod(a, b)`	Returns a pair `(a // b, a % b)` (quotient and remainder).

Examples

```python
print(abs(-10))       # 10
print(round(3.14159, 2)) # 3.14
# Note: round() rounds to the nearest even number for .5 cases
print(round(2.5))     # 2
print(round(3.5))     # 4

print(pow(2, 3))      # 8
print(divmod(10, 3))  # (3, 1) -> 3 with a remainder of 1
```

---

5. Object Introspection & Attributes

These functions help you examine and manipulate objects.

Function	What it does
`id(object)`	Returns the unique "identity" (memory address) of an object.
`type(object)`	Returns the type of an object.
`isinstance(object, classinfo)`	Returns `True` if the object is an instance of a class or a subclass.
`issubclass(class,`	Returns `True` if a class is a subclass of another class.

classinfo)	
dir(object)	Returns a list of valid attributes for the given object.
hasattr(object, name)	Returns `True` if an object has an attribute with the given name.
getattr(object, name)	Returns the value of an attribute of an object.
setattr(object, name, value)	Sets the value of an attribute on an object.
delattr(object, name)	Deletes an attribute from an object.

Examples

```
my_list = [1, 2]
print(id(my_list))      # e.g., 2328469384256
print(type(my_list))    # <class 'list'>

# isinstance() is preferred over type() for checking
print(isinstance(my_list, list)) # True
print(isinstance(my_list, (list, tuple))) # True

# Exploring an object
print(dir("hello")) # Shows all string methods like 'upper', 'lower', etc.

# Dynamic attribute access
class User:
    name = "Alice"

user = User()
print(hasattr(user, 'name'))  # True
print(getattr(user, 'name'))  # Alice
setattr(user, 'age', 30)
print(user.age)               # 30
delattr(user, 'name')
# print(user.name) # Would raise an AttributeError
```

Why & When to Use: Use `id()` and `is` for debugging object identity. Use `isinstance()` for robust type checking that respects inheritance. Use `dir()` for interactive exploration. Use `getattr`/`setattr` to write highly flexible code that can work with object attributes dynamically by name (as strings).

---

6. Working with Classes (OOP)

Function	What it does
super()	Returns a proxy object that delegates method calls to a parent or sibling class.
property()	Returns a property attribute (a more "Pythonic" way to create getters/setters).

`@classmethod`	A decorator to transform a method so it receives the class as the first argument, not the instance.
`@staticmethod`	A decorator to transform a method so it receives no special first argument (like a regular function inside a class).

Examples

```
class Person:
    def __init__(self, name):
        self.name = name

class Employee(Person):
    def __init__(self, name, employee_id):
        super().__init__(name) # Call parent's __init__
        self.employee_id = employee_id

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

    @property # This is a "getter"
    def radius(self):
        print("Getting radius")
        return self._radius

    @classmethod
    def from_diameter(cls, diameter):
        # A factory method that creates an instance from a diameter
        return cls(diameter / 2)

    @staticmethod
    def description():
        # Not tied to a specific instance or class
        return "A 2D shape with no corners."

c = Circle.from_diameter(10) # Creates a Circle with radius 5.0
print(c.radius)              # "Getting radius", then prints 5.0
print(Circle.description())  # "A 2D shape with no corners."
```

---

7. Utility & Other Functions

Function	What it does
`callable(object)`	Returns `True` if the object appears callable (i.e., can be called with `()`).
`hash(object)`	Returns the hash value of an object (if it has one).
`help(object)`	Invokes the built-in help system.
`ord(c)`	Given a string of one character, returns its Unicode code point.

`chr(i)`	Returns the string representing a character whose Unicode code point is the integer `i`.
`bin(x)`, `oct(x)`, `hex(x)`	Convert an integer to a binary, octal, or hexadecimal string.
`eval(expression)`	**DANGEROUS**. Parses and evaluates a string as a Python expression.
`exec(object)`	**DANGEROUS**. Executes Python code (which can be a string or object).

A Note on `eval()` and `exec()`

Warning: Never use `eval()` or `exec()` with untrusted input (e.g., from a user). They can execute arbitrary code, which is a massive security risk.

Examples

```
def my_func(): pass
print(callable(my_func)) # True
print(callable(123))     # False

# Hash is used for dict keys and sets
print(hash("hello"))     # A large integer
# print(hash([]))        # TypeError: unhashable type: 'list'

print(ord('A'))  # 65
print(chr(65))   # 'A'

print(bin(10))   # '0b1010'
print(hex(255))  # '0xff'
```

This list covers the vast majority of built-in functions you will encounter and use in day-to-day Python programming. Mastering them is a key step toward proficiency.