CSSE1001 — Week 7 Notes

Table of contents

Exceptions **2**
- Outline . . . 2
- Syntax errors vs. exceptions . . . 2
- Common run-time exceptions . . . 3
- Try/except . . . 4
- Catching specific vs. all exceptions . . . 4
- Exercise: robust input reading . . . 5
- General try/except framework . . . 6
- `else` and `finally` . . . 8
- Raising . . . 8
- Two exception-handling philosophies . . . 10
- Summary . . . 11

Introduction to Object-Oriented Programming **11**
- Outline . . . 11
- Programming paradigms . . . 11
- Object-oriented programming . . . 12
- I: Structure (holding named attributes) . . . 13
- II: Methods (object-scoped functions) . . . 15
- Exercise: a `Counter` object . . . 16
- Private variables . . . 17
- Setters . . . 17
- Summary . . . 19

Scope **19**
- Outline . . . 19
- Definition: scope . . . 20
- Global variables . . . 20
- Constants . . . 20
- Local variables shadow globals . . . 21

The `global` keyword . . . 21
A common gotcha: `UnboundLocalError` . . . 22
Shadowing . . . 23
Mixing globals and locals . . . 24
A `global` name doesn't exist until the function runs . . . 24
Python's LEGB rule for resolving names . . . 25
Worked: an invocation counter . . . 25
Summary . . . 26

Reference material **26**
Programming Paradigms . . . 26
Python Classes and Objects . . . 30
Python Exceptions . . . 32
Python Scope . . . 34

Exceptions

Lecture 7A. See csse1001[1] for course logistics and python-exceptions[2] for the full reference on `try`/`except` and raising errors.

Outline

- Syntax errors vs. run-time errors (exceptions)
- Common built-in exception types
- `try`/`except`, `else`, `finally`
- Catching specific vs. all exceptions
- Exercise: robust input reading
- Raising exceptions

Syntax errors vs. exceptions

When Python *parses* a file, it returns a **syntax error** if the contents don't correspond to valid Python code. Code that passes parsing transitions to **run-time**, where errors (**exceptions**) can occur — these differ from syntax errors because *you cannot detect them until they trigger*.

As a pipeline: `source → parse → (SyntaxError, or run-time) → exception → execution aborts`.

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-exceptions.pdf

Run-time errors are bad because they **abort the execution of the program** and all **intermediate work is lost** (unless saved to a file).

Common run-time exceptions

Exception	Description
`AssertionError`	an assertion fails
`IOError`	file does not exist
`IndexError`	index out of range
`KeyError`	key in dict does not exist
`NameError`	variable does not exist
`TypeError`	unexpected type is given to a function
`ValueError`	correct type, but inappropriate value
`ZeroDivisionError`	division by zero attempted

Four of them are shown below at the REPL. An uncaught exception prints a **traceback** before the error line; the traceback is shown in full once below and elided in the later transcripts.

```
>>> xs = [1, 2, 3]
>>> xs[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> xs = {'a': 1}
>>> xs['b']
KeyError: 'b'
```

`TypeError`'s message depends on which operand comes first, because it is the *left* operand's `+` that is tried first:

```
>>> "two" + 2
TypeError: can only concatenate str (not "int") to str
>>> 2 + "two"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The integer casting function has the type contract `int(xs: str) -> int`, and thus `int("ten")` should *not* throw a type error, because `"ten"` is indeed a string — it is the **value** that is wrong:

```
>>> int("ten")
ValueError: invalid literal for int() with base 10: 'ten'
```

Try/except

Catching an exception lets the program handle it gracefully.

```
try:
    <code>   # this code may throw an error
except ExceptionName:
    <code>   # this code is run when the error is of kind ExceptionName
```

(Also called *try-catch* in other languages, e.g. C++.) Typically used in user-facing layers of your software and/or when accessing/interacting with external resources — files, databases, API calls, network calls, and so on.

```
>>> x = 0
>>> 1/x               # without try-except
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> x = 0
>>> try:
...     1/x
... except ZeroDivisionError:      # it doesn't halt the execution
...     print("Please don't divide by zero...")
Please don't divide by zero
```

The `except ZeroDivisionError:` clause doesn't halt the execution. That is the whole point of the construct: the same division that aborted the program without it now just prints a message and carries on.

Catching specific vs. all exceptions

We can catch *any* exception without specifying the error type, but this can lead to undiscovered issues — it's better to catch specific error types:

```
>>> x = 0
>>> try:
...     print(y)
...     1/x
... except:
...     print("You did something fishy...")
```

It is **bad practice to ignore all errors**.

We can also catch among a list of exceptions:

```
>>> try:
...     print(y)     # this throws a name error
...     1/x
... except NameError:
...     print("You're using an undefined name...")
... except ZeroDivisionError:
...     print("You divided by zero...")
```

Exercise: robust input reading

Write a function

```
def io_double() -> int:
```

which takes no inputs, but rather prompts the user for a number and then **doubles** that number.

A first attempt:

```
>>> def io_double() -> int:
...     str_x = input("Number please: ")
...     int_x = int(str_x)
...     return 2*int_x

>>> io_double()
Number please: 2
4
>>> io_double()
Number please: two
    ...
ValueError: invalid literal for int() with base 10: 'two'
```

Wrapping the cast in a `try/except` inside a `while True:` loop lets the user retry until they succeed:

```
>>> def io_double() -> int:
...     while True:
...         str_x = input("Number please: ")
...         try:
...             int_x = int(str_x)
...             return 2*int_x   # unreachable if the line above errors
...         except ValueError:
...             print("That wasn't a number! Try again...")

>>> io_double()
Number please: two
That wasn't a number! Try again...
Number please: 3
6
```

The `return` line is unreachable if the cast above it errors. The `return` is inside the `try`, so if `int(str_x)` raises, control jumps to the `except` and the loop goes round again. Put the `return` outside the `try` and the retry behaviour disappears.

General try/except framework

```
try:
    <code>
except ExceptionName_0:
    <code>
except ExceptionName_1:
    <code>
    ...
except ExceptionName_k:
    <code>
```

The clauses are not a set — they are tried **in source order**, and the first one whose exception type matches wins. Figure 1 shows the search.

Python requires that **specific** exceptions appear before **general** ones:

```
try:
    <code>
```

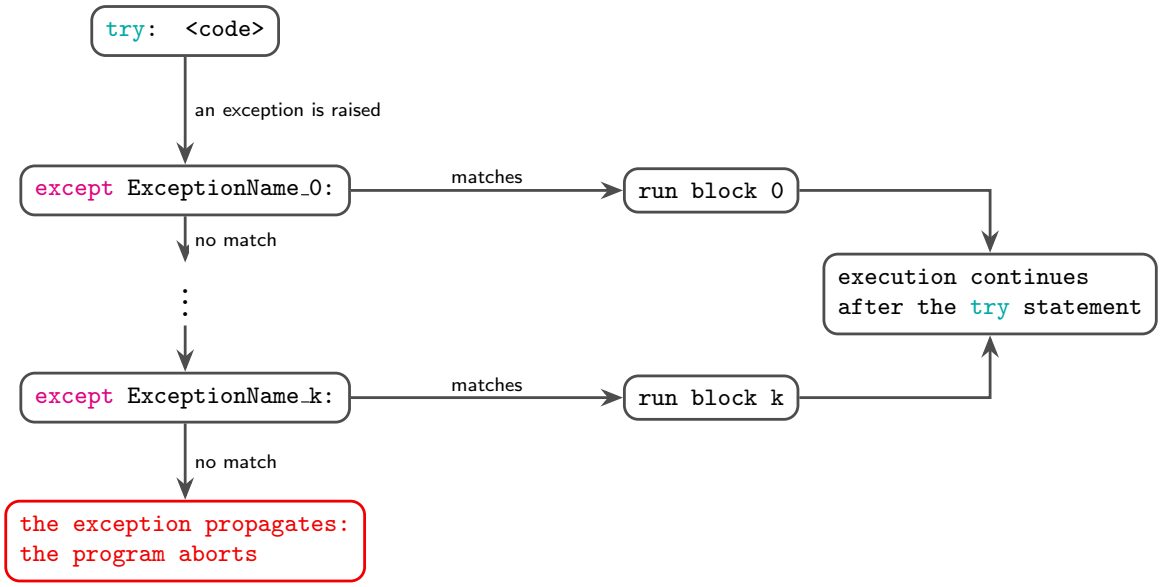

Figure 1: How a raised exception finds its handler: the except clauses are tried top to bottom, the first matching clause runs and the rest are skipped, and if none match the exception propagates and the program aborts. This is why the 'specific exceptions before general ones' rule matters — a general clause placed early swallows everything below it.

```
except ZeroDivisionError:
    <code>
except:
    <code>
```

`else` and `finally`

A `try` block can carry two further blocks beyond its `except` clauses. Figure 2 shows how the four fit together.

```
try:
    <code>       # run the code under try
except:
    <code>       # execute the code under except, when there is an exception
else:
    <code>       # no exceptions? run the code under else, after try
finally:
    <code>       # always run this code
```

```
try:
    x = int(input("Type a number: "))
    y = 1/x
    # note that if an exception is raised on a line of code here,
    # the following lines don't execute -- keep try blocks as short
    # as possible (don't put everything here!)
except ZeroDivisionError:
    print("Cannot enter zero")
except ValueError:
    print("Must type in a number")
else:
    print(y)
finally:
    print("this block gets always executed")
```

Raising

To **raise** an error (rather than catch it):

```
>>> def safe_div(x: int, y: int) -> float:
...     """
```

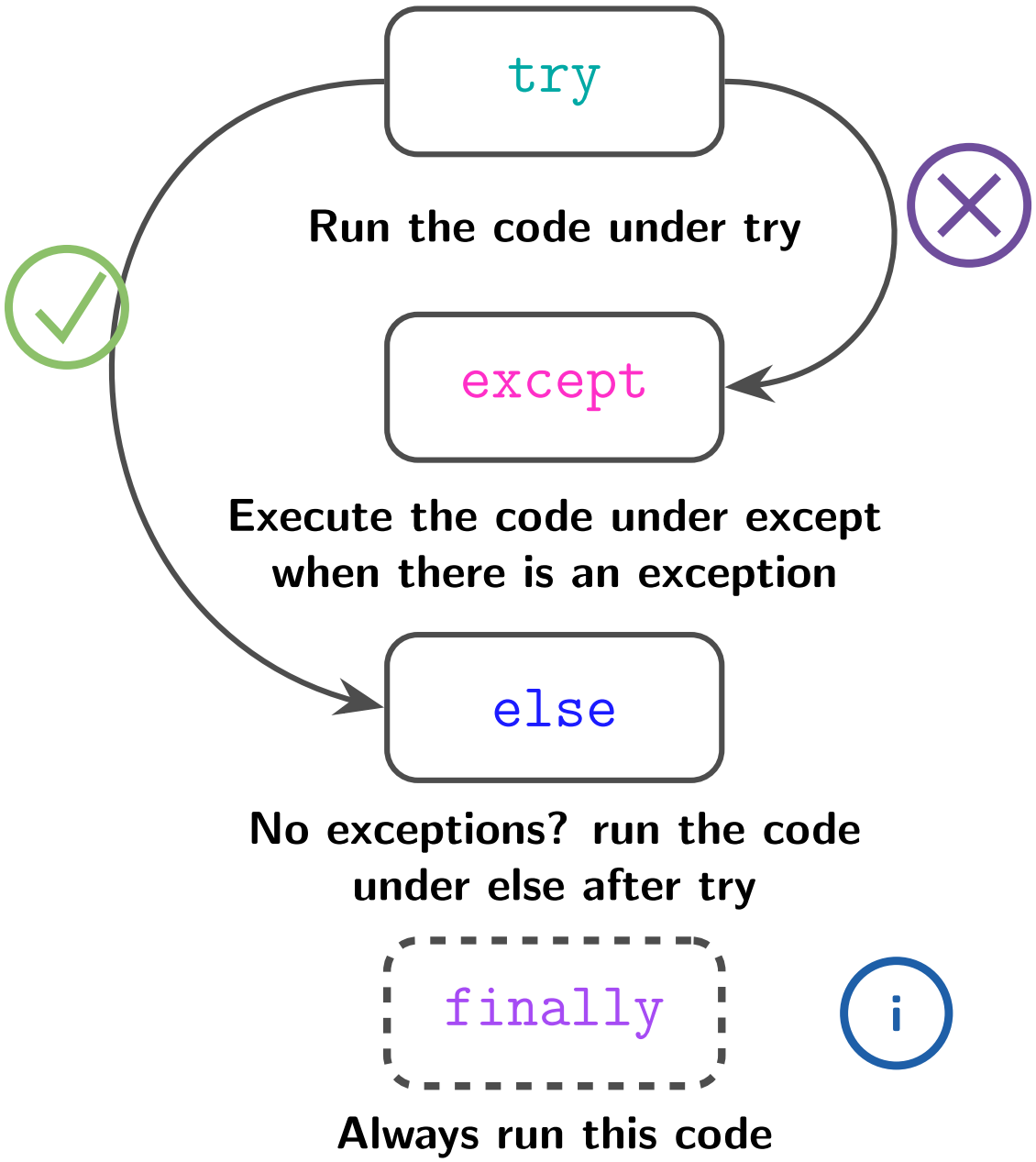

Figure 2: The four blocks associated with a try. The exception path leaves try on the right and lands in except (the purple crossed circle marks it); the no-exception path leaves on the left and skips straight past except into else (green tick). finally hangs below with no arrow into it at all, because it is not on either path — it runs either way, which is what the dashed border and the blue information glyph mark. Each keyword has its own colour: try teal, except magenta, else blue, finally purple.

```
...     Return 1 / (x-y)
...     """
...     if x == y:
...         raise ZeroDivisionError   # halt execution as soon as zero division
...     return 1/(x-y)

>>> safe_div(1, 1)
ZeroDivisionError
```

`raise ZeroDivisionError` halts execution as soon as the zero division is detected. `raise` is the opposite move to `try/except` — instead of absorbing a failure it manufactures one, at the earliest point where the failure is knowable, rather than letting a bad value travel further into the program.

```
def get_applicant_age() -> int:
    age = int(input("Enter your age: "))
    if age < 15:
        raise ValueError(f"Age must be >= 15 to apply for a license -- currently it's {age}")
    else:
        return age

try:
    get_applicant_age()
except ValueError:
    print("do something - don't issue license, etc")
```

Two exception-handling philosophies

LBYL (Look Before You Leap) — test first, then act:

```
if b == 0:
    result = None
    print("zero division!")
else:
    result = a/b
```

EAFP (Easier to Ask for Forgiveness than Permission) — act, then handle the failure:

```
try:
    result = a/b
```

```
except ZeroDivisionError:
    result = None
    print("zero division!")
```

Summary

We can **catch** errors which **throw** at **runtime** with the **try-except** control structure. We should only use a try-catch when **absolutely necessary**.

Next: 2025-09-09-scope[3] (Lecture 7B).

Introduction to Object-Oriented Programming

Lecture 7C. Course logistics are in csse1001[4]. The full reference on classes, instantiation, and encapsulation is python-classes-and-objects[5].

Outline

- Programming paradigms: imperative vs. declarative
- Objects, classes, and `self`
- Instantiation, attributes, equality, and aliasing
- Methods
- Private variables, getters, and setters

Programming paradigms

Imperative programming — the programmer *says* what to do, by:

1. **Procedural** — grouping instructions into functions.
2. **Object-oriented** — grouping instructions into objects that combine data (state, attributes) and behaviour (methods).

(*Imperative*, the adjective, means giving an authoritative command.)

Declarative programming — the programmer *declares* what they want, through:

1. **Functional** — a series of function applications.
2. **Logic** — a question about a system of facts and rules.

[3] courses/csse1001/2025-09-09-scope.pdf
[4] courses/csse1001/csse1001.pdf
[5] courses/csse1001/python-classes-and-objects.pdf

3. **Mathematical** — optimization.

In one line: **imperative** is *how* to do it, **declarative** is *what* to do. It is climbing rung by rung to the Moon versus simply floating there.

See programming-paradigms[6] for the full reference, including where each paradigm shows up across this course.

Object-oriented programming

The fundamental building block of object-oriented programming is the **class** or **object**. In this paradigm, the design principle is to solve a problem by creating objects that interact with one another.

> **Definition (Object).** An **object** is a collection of *fields* and *attributes* (i.e. data comprising the object's state) along with *methods* (i.e. class-scoped functions) that can act on the object itself. An object can *reference and change* its state, and has a notion of *self*.

Real-world analogues are in Figure 3. The split into a data half and a behaviour half is the whole idea.

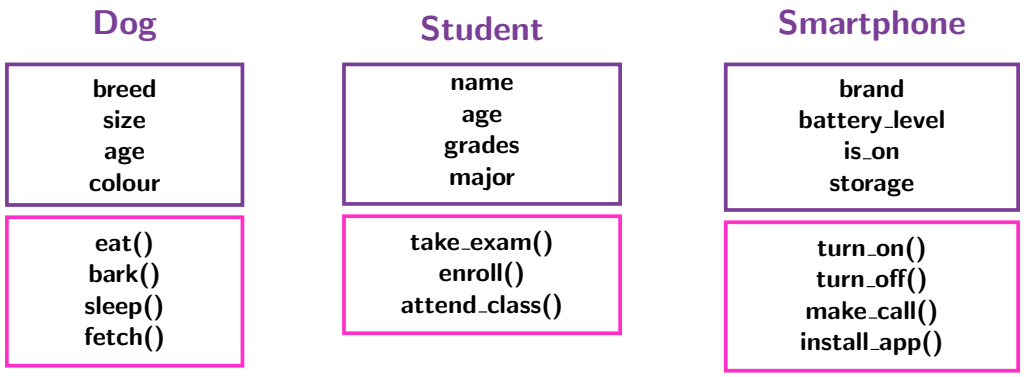

Figure 3: Three real-world classes. Each class is a name above two stacked boxes: the attributes (what it knows) on top and the methods (what it does) beneath. The attributes box is outlined in the dark purple used for class names and the methods box in bright magenta, so the two halves are told apart by outline colour, not just by position.

Convention: names of classes (i.e. objects) in Python are in `CamelCaps`. That is, `ClassNamesAreLikeThis`.

[6] courses/csse1001/programming-paradigms.pdf

I: Structure (holding named attributes)

```
class Point():                  # Point() is the class name
    def __init__(self):         # the initializer method
        self.x = 0              # -.
        self.y = 0              # -'  attributes
```

```
>>> p = Point()          # 'instantiation' of the point object
>>> p
<__main__.Point object at 0x10a9e0dd8>
>>> type(p)
<class '__main__.Point'>
>>> p.x
0
>>> p.y
0
```

`Point()` is the class name, `def __init__(self):` is the initializer method, and the two `self.` lines are the attributes. `p.x` accesses an attribute of an instance with `.`; these are called instance variables.

What is `self`?

Think of the class definition as a **blueprint** with placeholders or actual items. `self` has an `address`, `self` has `num_bedrooms`, `self` has `num_bathrooms`, `self` has a `garage`, `self` has `has_solar_panels` — these are the placeholders. When you create an *instance* of that class (e.g. `beach_house`), `self` becomes the actual object you're working with, and each of these becomes a real attribute stored in the object when it's created. You can create many instances using the same blueprint, each with its own unique values.

(The analogy is inspired by an example from MIT6.100L.)

```
>>> p = Point()
>>> p.x = 2
>>> p.y = 3
>>> p.x
2
>>> p.y
3
```

Initializing with input

Instantiating an object with input instead:

```
class Point():
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y
```

```
>>> p = Point(2, 3)   # creates a new object of type Point and passes 2 and 3 to __init__
>>> p.x
2
>>> p.y
3
```

Do not pass an argument for `self` — Python handles it automatically.

Equality and aliasing

Two separately-constructed objects with identical attribute values are **not `==`** by default; two *names for the same object* are. The difference is a difference in the picture of memory, not in the code that looks at it — see Figure 4.

```
>>> p = Point(2, 3)
>>> q = Point(2, 3)
>>> p == q
False
```

```
>>> p = Point(2, 3)
>>> q = p            # q is an alias for the same object as p
>>> q.x = 1
>>> p.x
1
>>> p == q           # because these aliases point to the same object
True
```

Equality (as opposed to identity) has to be defined explicitly — that is what `__eq__` is for, and it comes in the next lecture, 2025-09-16-dunder-methods[7].

[7] courses/csse1001/2025-09-16-dunder-methods.pdf

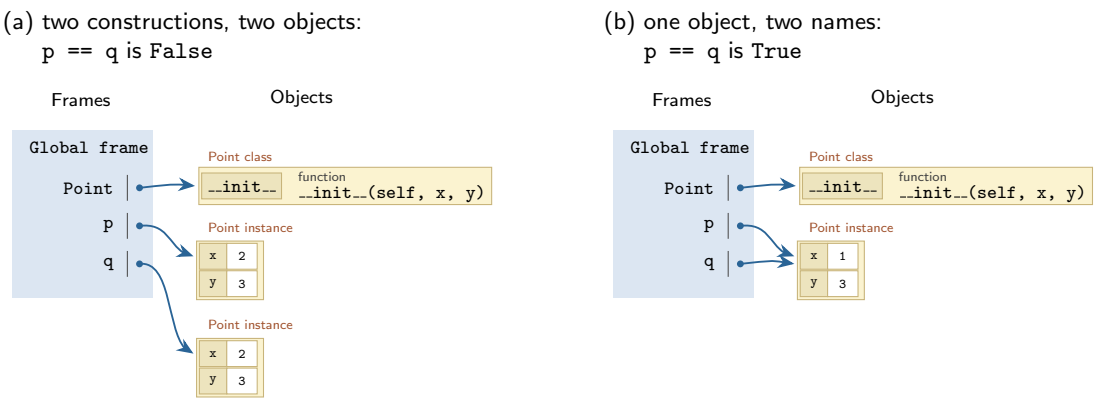

Figure 4: Equality and aliasing side by side, in Python Tutor notation. In (a) q = Point(2, 3) builds a second object, so p and q arrow off to two separate Point instances that happen to hold equal values, and p == q is False. In (b) q = p binds a second name to the one object, so both arrows converge on it — that convergence is the whole notation for aliasing — and mutating q.x is visible as p.x, and p == q is True. Both panels also show Point itself bound in the Global frame to the class object holding **init**, which is what makes Point(2, 3) callable at all.

II: Methods (object-scoped functions)

```
class Person():
    def __init__(self, name: str) -> None:   # a method for initializing
        self.name = name

    def foo(self) -> str:            # methods must take self as the first argument
        return f"My name is {self.name}."


>>> p = Person("Slim Shady")
>>> p.foo()
'My name is Slim Shady.'

>>> foo()                # you must access foo via the object
NameError: name 'foo' is not defined
```

A method is reached by **dot notation**: `<object_variable>.<method>(<parameters>)`.

```
>>> p = Person("What")
>>> q = Person("Who")
>>> r = Person(f"{2*'chka'} Slim Shady")
```

```
>>> p.foo()
'My name is What.'
>>> q.foo()
'My name is Who.'
>>> r.foo()
'My name is chka chka Slim Shady.'
```

Each instance keeps its own independent state.

> `f"{2*'chka'} Slim Shady"` actually evaluates to `"chkachka Slim Shady"` — for the output above the repeated string would need its own trailing space, `f"{2*'chka '}Slim Shady"`.

Exercise: a `Counter` object

Often our objects will be *analogous* to things that exist in the *real world*. Create an object called `Counter` that simulates the functionality of a hand-held tally counter.

```
class Counter():
    def __init__(self) -> None:
        self._value = 0            # a 'private' variable

    def get_value(self) -> int:    # this is called a 'getter'
        return self._value

    def click(self) -> None:
        self._value = self._value + 1

    def reset(self) -> None:
        self._value = 0
```

```
>>> x = Counter()
>>> x.get_value()
0
>>> x.click()
>>> x.click()
>>> x.click()
>>> x.get_value()
3
>>> x.reset()
>>> x.get_value()
0
```

Separate instances track their own count independently — each `Counter()` call builds its own object with its own `_value`, exactly as the two `Point` instances in Figure 4 (a) do:

```
>>> x = Counter()
>>> y = Counter()
>>> x.click()
>>> y.click()
>>> x.click()
>>> x.get_value()
2
>>> x.click()
>>> y.get_value()
1
```

Private variables

The underscore on `_value` in the `Counter` object is used to denote this name as **private**. This signals that programmers should *never* manipulate this value outside the object.

Warning: nothing actually *prevents* a user from accessing a private variable in Python!

```
>>> x = Counter()
>>> x.get_value()
0
>>> x.click()
>>> x._value = -10     # accessing private variables is bad practice
>>> x.click()
>>> x.get_value()
-9
```

Setters

It is good practice to use *methods* for changing an object's private variables at user level. We usually enable this through a `setter` method which *sets* a private value:

```
class Counter():
    def __init__(self) -> None:
        self._value = 0

    def set_value(self, x: int) -> None:   # a 'setter'
        if x < 0:
```

```
            raise ValueError
        self._value = x
```

The `raise` is the point of routing writes through a method: it is the only place the class gets to reject a value before it becomes state. (See python-exceptions[8].)

Classic getters and setters

```
# Classic implementation of getters and setters
class Person():
    def __init__(self, name):
        self._name = name   # leading underscore = "internal use"

    def get_name(self):
        return self._name

    def set_name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value

p = Person("Alice")
print(p.get_name())
p.set_name("Sara")
print(p.get_name())
```

Pythonic getters and setters

Python's `@property` decorator lets a getter/setter pair be used with plain attribute-access syntax, rather than explicit `get_`/`set_` method calls — the validation still runs, but the call sites read like ordinary attribute reads and writes:

```
# Pythonic implementation of getters and setters
class Person():
    def __init__(self, name):
        self._name = name

    @property
    def name(self):             # getter
```

[8] courses/csse1001/python-exceptions.pdf

```
        return self._name

    @name.setter
    def name(self, value):      # setter
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value

p = Person("Alice")
print(p.name)          # looks like attribute access (calls the getter)
p.name = "Sara"        # looks like assignment (calls the setter)
print(p.name)          # Sara
```

Summary

OOP helps organise code using **classes** and **objects**.

- **Class**: a blueprint for creating objects.
- **Object**: an instance of a class.
- **Encapsulation**: keep data safe inside classes using attributes and methods.

Next: magic methods — 2025-09-16-dunder-methods[9].

Scope

Lecture 7B. See csse1001[10] for course logistics and python-scope[11] for the full reference on global/local scope and the LEGB rule.

Outline

- Definition of scope
- Global variables and constants
- Local variables and shadowing
- The `global` keyword
- The LEGB name-resolution rule

[9] courses/csse1001/2025-09-16-dunder-methods.pdf
[10] courses/csse1001/csse1001.pdf
[11] courses/csse1001/python-scope.pdf

Definition: scope

The **scope** of a variable is the region of the code where the variable's name is recognized (i.e. the variable is accessible/visible).

```
>>> x = 2
>>> y = 3
>>> def foo():
...     return x
>>> def bar():
...     return foo()*y
>>> foo()
2
>>> bar()
6
```

`x` and `y` are **global** variables available to all functions. The global scope is the whole snippet. Figure 5 below draws global scope as a purple box.

Global variables

A **global variable** (or simply "global") is defined outside of any function (at the module level) and can be accessed by all functions in the module. Anything declared outside a function is globally accessible *provided there is no local variable with the same name*. A variable declared globally is said to have **global scope**.

Avoid global variables if possible: they can make code difficult to maintain.

Constants

By convention, **constants** are defined in `SNAKE_CASE_CAPS`:

```
PI = 3.14159
NUMBER_OF_DAYS_IN_WEEK = 7
GRID_SIZE = 5
```

Warning: unlike in other languages, the value of a constant is *not protected* and *can be changed at run-time*:

```
>>> PI = 3.14159
>>> PI = 3
>>> PI
3
```

Local variables shadow globals

```
>>> x = 2
>>> def foo():
...     x = 7   # local variable
...     return
>>> foo()
>>> x
2
```

Despite having the same name, the `x` of `foo()` is assumed **local** — its *scope* is `foo()`. Assigning to `x` inside the function creates a brand-new local variable rather than touching the global one, which is why `x` is still `2` after the call.

Figure 5 draws this. A rounded box over a span of code lines *is* a scope, and the box's colour names the kind of scope.

The `global` keyword

We can specify that `foo` should be using `x` as a global. It is good practice to declare your globals when you use one:

```
>>> x = 2
>>> def foo():
...     global x
...     x = 7
...     return
>>> foo()
>>> x
7
```

If a function only ever *creates* a local variable (no `global` declaration), that name doesn't exist outside the function — outside of `foo` the variable `x` does not exist; `x` is a **local variable**:

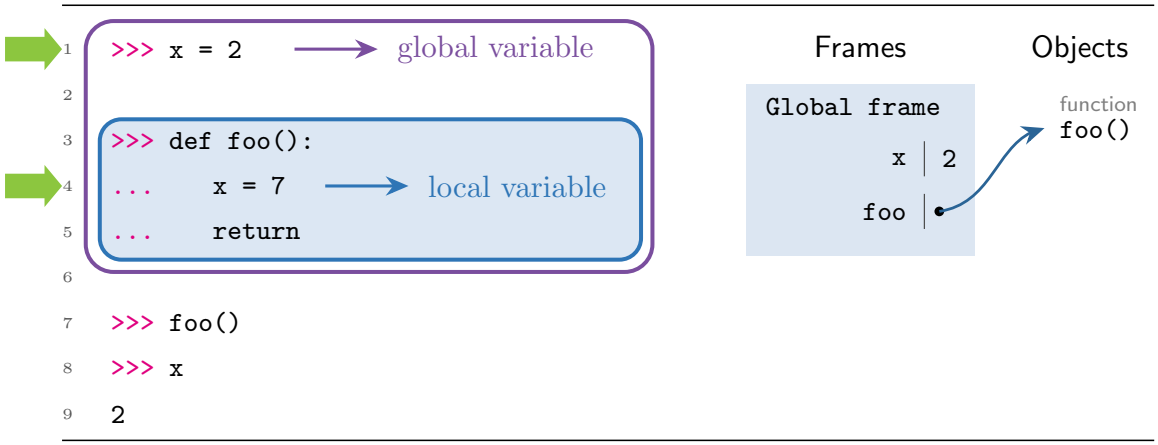

Figure 5: Scope boxes over a snippet. The purple outline is global scope and encloses the whole snippet; the blue outline with a pale blue fill is local scope and encloses only the body of the def, nested inside the purple box. Each box has a short arrow out to a colour-matched label. The green markers in the far-left margin flag lines 1 and 4 — the two assignments to x. Beside the code is the Python Tutor memory picture: the Global frame holds x bound to 2 and foo bound (blue arrow) to a function object, and there is no trace of foo's x at all, because the local x died when the call returned.

```
>>> def foo():
...     x = 2       # local variable
...     return x
>>> foo()
2
>>> x
NameError: name 'x' is not defined
```

A common gotcha: `UnboundLocalError`

```
>>> x = 2
>>> def foo():
...     x = x + 2
...     return
>>> foo()
UnboundLocalError: local variable 'x' referenced before assignment.
```

When `foo` creates `x`, it becomes local and thereby has no value at the time of assignment. As soon as `foo` assigns to `x` anywhere in its body, Python treats `x` as local *throughout the whole*

function — so the right-hand side `x + 2` tries to read a local `x` that doesn't have a value yet. Declaring `global x` first fixes this, since `x` then refers to the module-level variable throughout, and the effect accumulates across calls:

```
>>> x = 2
>>> def foo():
...     global x
...     x = x + 2
...     return
>>> foo()
>>> x
4
>>> foo()
>>> x
6
```

Shadowing

A *parameter* is the other way a local name comes into existence, and it shadows a global just as an assignment does. Figure 6 applies the same scope boxes to that case.

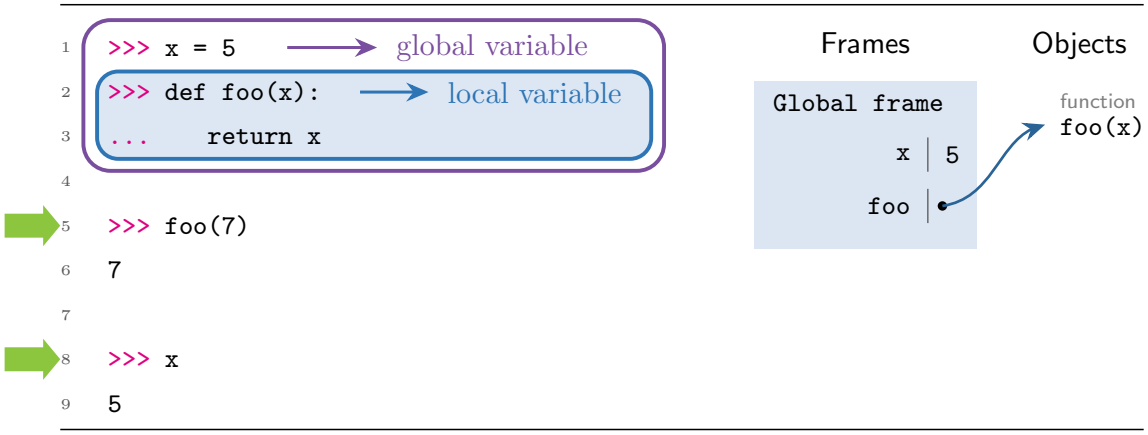

Figure 6: Shadowing by a parameter, in the same notation as Figure 5. The only structural change from that figure is where the blue local box starts: it opens on the def line itself, because here the local x is the parameter rather than an assignment in the body. foo(7) returns 7 while the global x is still 5 — the green margin markers flag exactly those two lines — and the Python Tutor panel shows why: the only x anywhere in the Global frame is the global one, bound to 5.

Despite having the same name, there are two `x`'s: one with a global scope and another with local. The parameter **shadows** the global variable, making it temporarily invisible inside `foo`.

A function can't declare a name as both a parameter and `global` at the same time:

```
>>> x = 5
>>> def foo(x):
...     global x
...     return
SyntaxError: name 'x' is parameter and global
```

Mixing globals and locals

```
>>> x = 5
>>> def foo(y):
...     return x*y
>>> foo(7)
35
>>> foo(x)
25
```

Globals and locals can be used in mixed computation.

A `global` name doesn't exist until the function runs

Declaring `global x` inside a function does not, by itself, create the module-level `x`. The binding only happens when the assignment actually executes — i.e. when the function is *invoked*:

```
>>> def foo():
...     global x
...     x = 2
...     return

>>> x
NameError:

>>> foo()
>>> x
2
```

Python's LEGB rule for resolving names

When Python looks up a name it searches four namespaces in a fixed order, dropping to the next only when the name is not found — and raising an error if it runs out. Figure 7 shows the cascade.

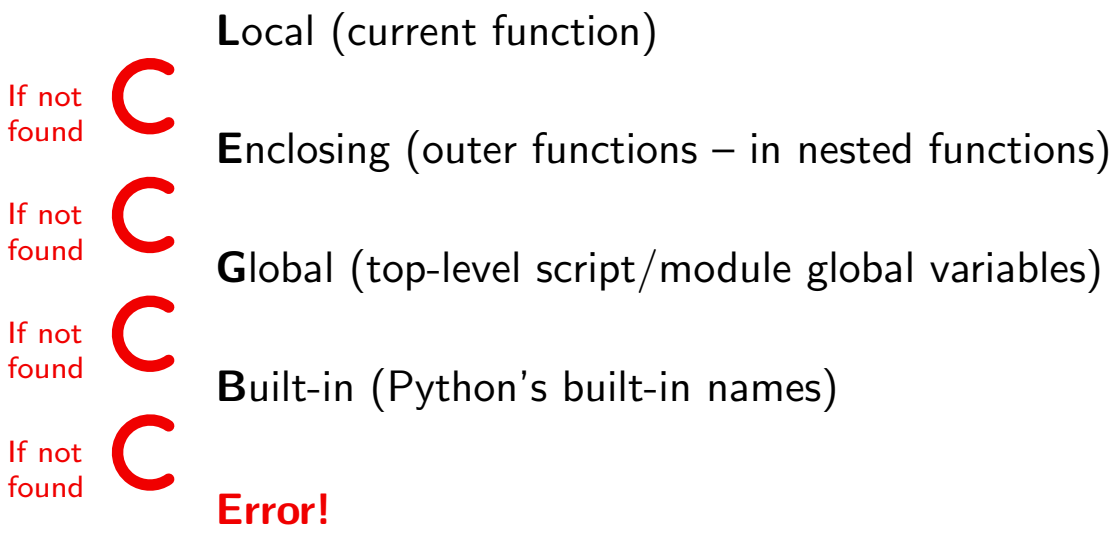

Figure 7: Python's LEGB rule. Read it top to bottom: each red hook is one 'if not found' step down to the next enclosing namespace, and the cascade bottoms out in a red Error! rather than in a default value. The bolded initial letters spell the mnemonic, and red marks the failure path.

1. **L**ocal — the current function.
2. **E**nclosing — outer functions, for nested functions.
3. **G**lobal — top-level script/module global variables.
4. **B**uilt-in — Python's built-in names.

If the name isn't found at any level: **Error!**

Worked: an invocation counter

Write a function `foo` that returns the number of times the function has been called/invoked. For example:

```
>>> foo()
1
```

```
>>> foo()
2
>>> foo()
3
```

A solution with the `global` keyword:

```
count = 0

def foo() -> int:
    """ Return the number of times foo() has been called. """
    global count
    count += 1
    return count
```

The state has to live *outside* the function: a `count = 0` inside `foo` would be re-created local and reset to `0` on every call, and `count += 1` without the `global` declaration is exactly the `UnboundLocalError` case above. `print(count)` in place of `return count` would produce the identical REPL display — the REPL doesn't echo a `None` return — while giving a function that returns nothing, so it would not satisfy the task as stated.

Summary

The places in your program that can access a **name** is called the **scope** of that name. The **global scope** is for things like **constants**, whereas names **declared in functions** get **locally scoped** to that function.

Next: 2025-09-09-object-oriented-programming[12] (Lecture 7C).

Reference material

Programming Paradigms

A **paradigm** is the style in which you describe a computation to a machine. The top-level split is between saying *how* to do something and saying *what* you want.

	Question answered
Imperative	How to do it

[12] courses/csse1001/2025-09-09-object-oriented-programming.pdf

	Question answered
Declarative	What to do

Imperative programming

The programmer spells out *how* to do it, step by step. (Imperative, as an adjective, means giving an authoritative command.) It divides further by how those commands are grouped:

1. **Procedural** — groups instructions into *functions*. This is everything up to week 7: python-functions[13], python-if-statements[14], python-while-loops[15], python-for-loops[16].
2. **Object-oriented** — groups instructions into *objects* that combine data (state, attributes) and behaviour (methods). This is weeks 7–11: python-classes-and-objects[17] onward.

Both are imperative — OOP does not stop you writing step-by-step instructions, it just changes what those instructions are attached to.

Declarative programming

The programmer declares what they want, through:

1. **Functional** — a series of function applications.
2. **Logic** — a question asked about a system of facts and rules.
3. **Mathematical optimization** — an objective and a set of constraints.

Python is not a declarative language, but it has declarative *moments*. A list comprehension is the clearest one:

```
# imperative: how to build the list
acc = []
for k in range(4):
    acc.append(k**2)

# declarative: what the list is
[k**2 for k in range(4)]
```

The comprehension reads like the set-builder notation $\{k^2 : k \in \{0, 1, 2, 3\}\}$ — you state the set, not the procedure for accumulating it. See python-comprehensions[18].

[13] courses/csse1001/python-functions.pdf
[14] courses/csse1001/python-if-statements.pdf
[15] courses/csse1001/python-while-loops.pdf
[16] courses/csse1001/python-for-loops.pdf
[17] courses/csse1001/python-classes-and-objects.pdf
[18] courses/csse1001/python-comprehensions.pdf

Paradigm is not the same as level of abstraction

Take **one program written six times over** — integer exponentiation, `power(x, y)` — in six languages, from raw x86 bytes up to Python (the listings themselves are walked through in history-of-computing[19]). It is tempting to read that ladder as a tour of paradigms. It is not. Every rung is **imperative**: set an accumulator to 1, run a loop `y` times, multiply.

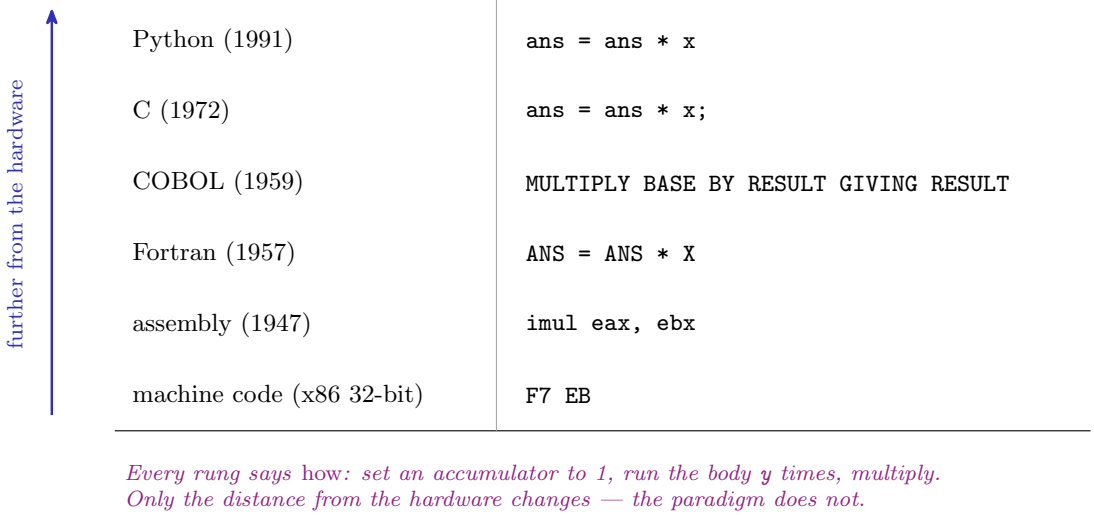

Figure 8: The one step that survives all six languages. Each row is that language's way of writing *multiply the accumulator by x*. The rows run from the oldest language at the bottom to the newest at the top. The bottom two rows are the same instruction twice: `F7 EB` is what the machine stores, `imul eax, ebx` is the assembly for it.

The loop is the clearest tell. All six spell out *how* to repeat:

Language	How it repeats
machine code	`74 09` and `EB F5` — a conditional jump 9 bytes forward, an unconditional jump 11 bytes back
assembly	`test ecx, ecx` / `jz .done` … `jmp .loop`
Fortran	`DO 10 I = 1, Y` … `10 CONTINUE`
COBOL	`PERFORM UNTIL COUNTER = 0` … `END-PERFORM.`

[19] courses/csse1001/history-of-computing.pdf

Language	How it repeats
C	`for (int i = 0; i < y; i++)`
Python	`for _ in range(y)`

COBOL looks least like the rest. `MULTIPLY BASE BY RESULT GIVING RESULT` is English, and that was the point: Grace Hopper's contribution was *machine independent programming based on english words*. But English-like syntax is not a declarative paradigm — the COBOL program still names every step and every repetition, in order. It is imperative prose.

So the ladder measures a different axis from the imperative/declarative split:

- **Level of abstraction** — how much of the machine you have to say out loud. This is what changes from `F7 EB` to `ans = ans * x`.
- **Paradigm** — whether you say *how* or *what*. This does not change anywhere on the ladder.

Python's `**` operator (see 2025-07-29-python-as-a-calculator[20]) needs no loop written at all, and is one more step up the *first* axis: `x ** y` hands the repetition to the language. That still is not a change of paradigm — a single operator call is an abstraction, not a style of description. The genuine paradigm shift is the one shown further up, where a comprehension replaces the accumulation with a statement of what the result *is*. For `**` alongside the rest of the arithmetic, see python-operator-precedence-and-associativity[21].

Why Python has several

Python was created to *support a variety of programming styles* rather than enforce one (see history-of-computing[22]). CSSE1001 uses this deliberately: the course is structured as procedural first, then object-oriented, then (in the final, non-assessed week) a look at the functional style.

Where OOP fits

> The fundamental building block of object-oriented programming is the class or object. In this paradigm, the design principle is to solve a problem by creating objects that interact with one another.

[20] courses/csse1001/2025-07-29-python-as-a-calculator.pdf
[21] courses/csse1001/python-operator-precedence-and-associativity.pdf
[22] courses/csse1001/history-of-computing.pdf

An **object** is a collection of fields and attributes (the data comprising the object's state) along with methods (class-scoped functions) that can act on the object itself. An object can reference and change its own state, and has a notion of `self`.

Continue in python-classes-and-objects[23] for the mechanics, and 2025-09-09-object-oriented-programming[24] for the lecture that introduces them.

Python Classes and Objects

An **object** bundles data (**fields/attributes**, comprising its state) with **methods** (class-scoped functions that act on that data). An object can reference and change its own state, and so has a notion of *self*.

A **class** is the blueprint; an **object** is an instance of that class. Keeping the data inside the class and reaching it only through the class's own attributes and methods is **encapsulation**.

Introduced in 2025-09-09-object-oriented-programming[25] (Lecture 7C).

Defining a class

```
class ClassName():
    def __init__(self, ...):
        self.attribute = ...

    def some_method(self, ...):
        return ...
```

- Class names use `CamelCaps` by convention (`ClassNamesAreLikeThis`).
- `__init__` is the **initializer**, run automatically when an instance is created.
- Every method's first parameter is `self` — Python supplies it automatically; never pass an argument for it explicitly.
- Think of the class body as a blueprint with placeholders: `self` has an `x`, `self` has a `y`. On **instantiation**, `self` becomes the particular object being built and each placeholder becomes a real attribute stored in it, so many instances can share one blueprint while each holds its own values.

[23] courses/csse1001/python-classes-and-objects.pdf
[24] courses/csse1001/2025-09-09-object-oriented-programming.pdf
[25] courses/csse1001/2025-09-09-object-oriented-programming.pdf

Instantiation and attributes

```
>>> p = ClassName(...)       # creates an instance, passing args to __init__
>>> p.attribute              # dot notation accesses instance variables
>>> p
<__main__.ClassName object at 0x10a9e0dd8>
>>> type(p)
<class '__main__.ClassName'>
```

A method is called the same way: `<object_variable>.<method>(<parameters>)`. Calling it as a bare name is a `NameError` — it only exists on the object.

Equality vs. aliasing

- Two separately-constructed instances with identical attributes are **not** `==` by default: the default comparison is identity, so equal-looking objects at two different addresses compare `False`. Defining what equality *means* for a class is the job of `__eq__` — see python-dunder-methods[26].
- Assigning `q = p` makes `q` an **alias** for the same object as `p` — mutating one is visible through the other, and `p == q` holds because they're literally the same object.

Private variables, getters, and setters

A leading underscore (`self._value`) signals that an attribute is intended for internal use only — a *convention*, not an enforced restriction; nothing stops external code from reading or writing it directly (which is bad practice).

Prefer exposing controlled access via methods:

```
def get_value(self) -> int:            # getter
    return self._value

def set_value(self, x: int) -> None:   # setter
    if x < 0:
        raise ValueError
    self._value = x
```

[26] courses/csse1001/python-dunder-methods.pdf

The `raise` is the reason the setter exists: it is the one place the class can reject a bad value before it becomes state (see python-exceptions[27]).

Python's `@property`/`@<name>.setter` decorators let a getter/setter pair be used with ordinary attribute syntax (`obj.name` / `obj.name = value`) instead of explicit method calls, so validation still runs while call sites stay readable.

Python Exceptions

An **exception** is a run-time error — unlike a **syntax error**, which Python reports while *parsing* the file, an exception cannot be detected until the offending line actually executes. Left uncaught it **aborts the program**, and all intermediate work is lost unless it was already written to a file.

Introduced in 2025-09-09-exceptions[28] (Lecture 7A).

Common built-in exceptions

Exception	Raised when…
`AssertionError`	an `assert` fails
`IOError`	a file does not exist
`IndexError`	a sequence index is out of range
`KeyError`	a dict key does not exist
`NameError`	a variable/name does not exist
`TypeError`	an unexpected type is given to a function/operator
`ValueError`	correct type, but an inappropriate value
`ZeroDivisionError`	division by zero

The `TypeError`/`ValueError` boundary follows the function's **type contract**: `int` has the contract `int(xs: str) -> int`, so `int("ten")` is given the right *type* and therefore raises `ValueError`, not `TypeError`.

Catching exceptions

```
try:
    <code that may raise>
except SomeError:
    <handle SomeError>
```

[27] courses/csse1001/python-exceptions.pdf
[28] courses/csse1001/2025-09-09-exceptions.pdf

```
except AnotherError:
    <handle AnotherError>
else:
    <runs only if no exception occurred>
finally:
    <always runs>
```

Also called **try-catch** in other languages (e.g. C++). Typically used in the user-facing layers of a program, and wherever it touches external resources — files, databases, API calls, network calls.

- The `except` clauses are tried **in source order** and the **first match wins**; the rest are skipped. If none match, the exception propagates and the program aborts.
- Consequently, **specific exceptions must be listed before general ones** (and before a bare `except:`) — Python enforces this ordering, because a general clause placed first would swallow every case below it.
- A bare `except:` catches *any* exception, but hides genuine bugs — prefer catching specific exception types. It is bad practice to ignore all errors.
- `else` runs only when the `try` block completed without raising; `finally` runs on every path, exception or not, and is the right home for cleanup.
- Keep `try` blocks short: once an exception is raised inside one, none of the remaining lines in that block execute — so a `return` placed after the risky line will simply be skipped.

Raising exceptions

```
raise SomeError                  # bare
raise SomeError("message")       # with an explanatory message
```

`raise` halts execution at the point the failure becomes knowable, rather than letting a bad value travel further into the program. It is the standard way for a **setter** to reject an invalid value before it becomes an object's state — see python-classes-and-objects[29].

LBYL vs. EAFP

Two philosophies for guarding against errors:

- **LBYL** (Look Before You Leap) — check the condition with an `if` before attempting the risky operation.
- **EAFP** (Easier to Ask for Forgiveness than Permission) — attempt the operation inside a `try`, and handle the resulting exception if it fails.

[29] courses/csse1001/python-classes-and-objects.pdf

Only use a `try/except` when it is *absolutely necessary*: an exception handler that is really a misplaced `if` makes the ordinary path harder to read.

Python Scope

The **scope** of a name is the region of code where that name is recognized (i.e. accessible/visible). Python resolves names using the **LEGB** rule, searching in order:

1. **L**ocal — the current function.
2. **E**nclosing — an outer function, for nested functions.
3. **G**lobal — the module's top-level variables.
4. **B**uilt-in — Python's built-in names.

If a name isn't found at any level, Python raises a `NameError`.

Introduced in 2025-09-09-scope[30] (Lecture 7B), which also draws the LEGB cascade and the course's global/local scope boxes.

Global variables

Defined outside any function (module level); accessible everywhere in the module *provided* no local variable shadows the name. Avoid globals where possible — they make code harder to maintain.

Constants are conventionally named in `SNAKE_CASE_CAPS` (`PI`, `GRID_SIZE`, `NUMBER_OF_DAYS_IN_WEEK`), but unlike in other languages Python does **not** protect their value: a constant can simply be reassigned at run-time.

Local variables and shadowing

Assigning to a name anywhere inside a function body makes that name **local to the entire function** — even before the assignment line executes. So does taking it as a **parameter**. This causes two common gotchas:

- **Shadowing**: a local variable (or parameter) with the same name as a global temporarily hides the global inside that function. The global itself is untouched — it still holds its old value after the call returns.
- `UnboundLocalError`: if a function reads a name before assigning to it, and also assigns to that same name later in its body, Python treats it as local throughout — so the read fails, because the local doesn't have a value yet.

[30] courses/csse1001/2025-09-09-scope.pdf

The `global` keyword

Declares that a name inside a function refers to the module-level variable, rather than creating a local shadow:

```
def foo():
    global x
    x = x + 1
```

- It is good practice to declare a global explicitly wherever a function writes to one.
- A name cannot be declared as both a function parameter and `global` in the same function — that is a `SyntaxError` ("name 'x' is parameter and global").
- Declaring `global x` does not by itself create `x` at module level. The name only comes into existence when the assignment actually runs, i.e. when the function is **invoked** — until then, reading `x` at the prompt is still a `NameError`.
- Globals and locals mix freely in one expression: `def foo(y): return x*y` reads `x` globally and `y` locally.