CSSE1001 — Week 7 Notes

Table of contents

Exceptions

See csse1001[1] for course logistics — this note covers Lecture 7A's technical content. See python-exceptions[2] for the full reference on `try`/`except` and raising errors.

Today's 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 they can't be detected until they trigger.

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

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

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

```
>>> xs = [1, 2, 3]
>>> xs[4]
IndexError: list index out of range

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

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

Note that `int(xs: str) -> int`'s type contract means `int("ten")` shouldn't raise a *type* error, because `"ten"` is indeed a string — it's the *value* that's wrong:

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

Try/except

```
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 software and/or when accessing/interacting with external resources (files, databases, API calls, network calls, etc).

```
>>> x = 0
>>> 1/x               # without try-except
ZeroDivisionError: division by zero
```

```
>>> x = 0
>>> try:
...     1/x
... except ZeroDivisionError:
...     print("Please don't divide by zero...")
Please don't divide by zero
```

It doesn't halt execution.

Catching specific vs. all exceptions

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

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

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 prompts the user for a number and doubles it.

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 casting 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
```

General try/except framework

```
try:
```

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

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

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

`else` and `finally`

```
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:
...     """ 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
```

```
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):

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

EAFP (Easier to Ask for Forgiveness than Permission):

```
try:
    result = a/b
except ZeroDivisionError:
    result = None
    print("zero division!")
```

Summary

We can catch errors that are thrown at run-time with the `try/except` control structure. We should only use a `try/except` when absolutely necessary.

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

Introduction to Object-Oriented Programming

See csse1001[4] for course logistics — this note covers Lecture 7C's technical content. See python-classes-and-objects[5] for the full reference on classes, instantiation, and encapsulation.

Today's 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 *how* to do something, 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 says *what* they want, through:

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

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

Object-oriented programming

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

An **object** is a collection of fields/attributes (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**.

Real-world analogues:

Class	Attributes	Methods
`Dog`	`breed`, `size`, `age`, `colour`	`eat()`, `bark()`, `sleep()`, `fetch()`
`Student`	`name`, `age`, `grades`, `major`	`take_exam()`, `enroll()`, `attend_class()`
`Smartphone`	`brand`, `battery_level`, `is_on`, `storage`	`turn_on()`, `turn_off()`, `make_call()`, `install_app()`

Convention: class names in Python are in `CamelCaps` — `ClassNamesAreLikeThis`.

I: Structure (holding named attributes)

```
class Point():
    def __init__(self):
        self.x = 0
        self.y = 0


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

```
0
```

Attributes of an instance are accessed with `.` — these are called **instance variables**.

What is `self`?

Think of the class definition as a *blueprint* with placeholders. `self` has an `x`, `self` has a `y`, and so on — these are the placeholders. When you create an *instance* of the class (e.g. `p`), `self` becomes the actual object you're working with, and each placeholder becomes a real attribute stored in that object. You can create many instances from the same blueprint, each with its own unique values.

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

Initializing with input

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

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

Careful! Don't pass an argument for `self` — Python supplies it automatically.

Equality

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

Two separately-constructed objects with identical attribute values are **not** `==` by default — equality (as opposed to identity) needs to be defined explicitly (covered in a later lecture on magic methods).

Aliasing

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

Because `q` and `p` are aliases pointing to the *same* object, mutating `q` also changes what `p` sees, and (since it's literally the same object) `p == q` is `True` here.

II: Methods (object-scoped functions)

```
class Person():
    def __init__(self, name: str) -> None:
        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()
NameError: name 'foo' is not defined
```

A method must be accessed via the object using 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.

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-tally counter.

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

    def get_value(self) -> int:    # 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:

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

Private variables

The leading underscore on `_value` in `Counter` signals that this name is **private** — programmers should never manipulate it directly from outside the object.

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

```
>>> x = Counter()
>>> x.click()
>>> x._value = -10
>>> x.click()
>>> x.get_value()
-9
```

Accessing private variables directly is bad practice.

Setters

It's good practice to use a method — a **setter** — for changing an object's private variables at the user level, so that invalid values can be rejected:

```
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
```

Classic 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:

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

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

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.

Scope

See csse1001[6] for course logistics — this note covers Lecture 7B's technical content. See python-scope[7] for the full reference on global/local scope and the LEGB rule.

Today's outline

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

[6] courses/csse1001/csse1001.pdf
[7] 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.)

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` inside `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.

The `global` keyword

We can specify that a function should use a name as a global (rather than create a local shadow). It's 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:

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

A common gotcha: `UnboundLocalError`

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

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

Shadowing

```
>>> x = 5
>>> def foo(x):
...     return x
>>> foo(7)
7
>>> x
5
```

Despite having the same name, there are two `x`'s: one with global scope, and another local to `foo` (its parameter). 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 freely used together in the same expression.

Python's LEGB rule for resolving names

When Python looks up a name, it searches (in order):

1. **L**ocal — the current function.
2. **E**nclosing — outer function(s), 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, Python raises an error.

Exercise: an invocation counter

Write a function `foo` that returns the number of times it has been called:

```
>>> foo()
1
>>> foo()
2
>>> foo()
3
```

```
count = 0
def foo():
    """ prints the number of times foo() has been called """
    global count
    count += 1
    print(count)
```

> The task asks for a function that *returns* the count, but the source's implementation only `prints` it (returning `None` implicitly) — this happens to produce identical output in the REPL shown above, since a `None` return isn't echoed, but the two aren't the same thing. Reproduced here exactly as given, with the discrepancy flagged rather than silently "fixed" into a `return count`.

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[8] (Lecture 7C).

Reference material

Python Classes and Objects

An **object** bundles data (**attributes**, comprising its state) with **methods** (functions that act on that data) — the object has a notion of *self*.

Defining a class

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

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

- Class names use `CamelCaps` by convention.
- `__init__` is the **initializer**, run automatically when an instance is created.

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

- Every method's first parameter is `self` — Python supplies it automatically; never pass an argument for it explicitly.

Instantiation and attributes

```
>>> p = ClassName(...)      # creates an instance, passing args to __init__
>>> p.attribute              # dot notation accesses instance variables
```

Equality vs. aliasing

- Two separately-constructed instances with identical attributes are **not** `==` by default (equality compares identity unless a class defines otherwise).
- 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
```

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.

Encapsulation

Encapsulation means keeping an object's data safe inside the class, exposed only through its attributes and methods.

Python Exceptions

An **exception** is a run-time error — unlike a syntax error, it can't be detected until the offending line actually executes. Left uncaught, it aborts the program.

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

Catching exceptions

```
try:
    <code that may raise>
except SomeError:
    <handle SomeError>
except AnotherError:
    <handle AnotherError>
else:
    <runs only if no exception occurred>
finally:
    <always runs>
```

- **Specific exceptions must be listed before a bare `except:`** — Python enforces this ordering.
- A bare `except:` catches *any* exception, but hides genuine bugs — prefer catching specific exception types.
- Keep `try` blocks short: once an exception is raised inside one, the rest of the block is skipped.

Raising exceptions

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

LBYL vs. EAFP

Two equally valid 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.

Python Scope

The **scope** of a name is the region of code where that name is recognized. 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`.

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`, but Python does not actually protect their value from being reassigned.

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. 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.

- `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.

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
```

A name cannot be declared as both a function parameter and `global` in the same function — this is a `SyntaxError`.