# Exceptions and Files

Colan Biemer

# Exceptions

# Errors We Have Seen

| Exception | What caused it | Where |
|---|---|---|
| `NameError` | calling `say_hello()` above its `def` | Define Before You Call |
| `AttributeError` | calling `append` on `None` | `append` Gives Back `None` |
| `IndexError` | `scores[5]` on a list of five | Index Out of Range |

And more... Each of these is an **exception**: Python stops the program and names what went wrong. Instead of avoiding them, though, we can `catch` them.

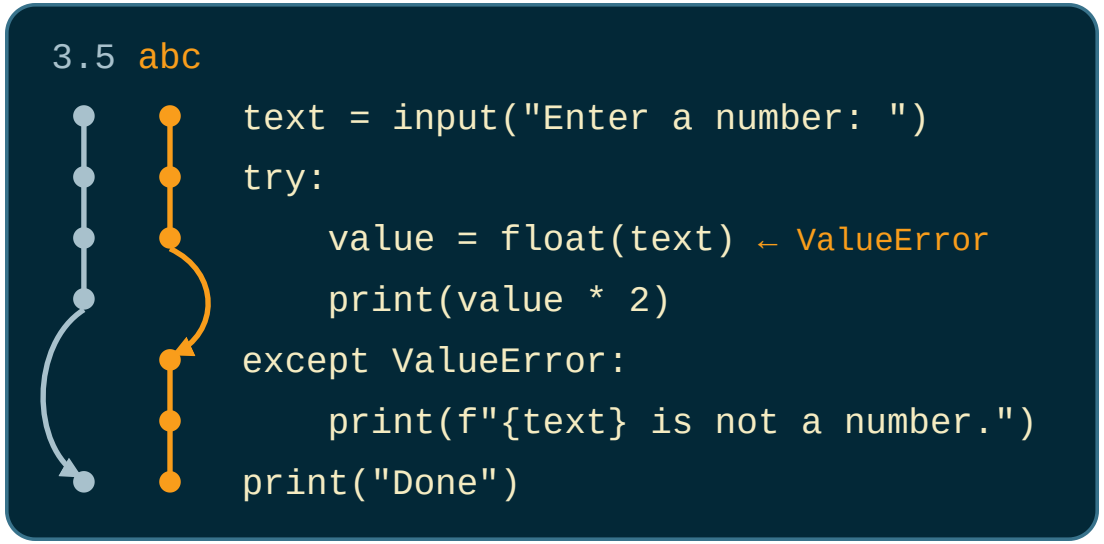
## New Keywords: `try` and `except`

```
try:
    # code that might raise an exception
except [EXCEPTION]:
    # runs only if the code in try raised EXCEPTION

# program continues as normal
```

Notice the colons and the indentation, just like `if` and `else`.

# Catching a `ValueError`




```
Enter a number: 3.5
7.0
Done
```

```
Enter a number: abc
abc is not a number.
Done
```

With `3.5`, every line in `try` runs, and Python skips the `except` block. With `abc`, `float(text)` raises a `ValueError`. Python stops the `try` block on that line, jumps to `except ValueError:`, and runs its body. Either way, the program reaches `print("Done")`.

# Problem: What Is the Output?

```
try:
    print("A")
    n = int("seven")
    print("B")
except ValueError:
    print("C")
print("D")
```

## Solution: What Is the Output?

```
try:
    print("A")          # A
    n = int("seven")    # ValueError
    print("B")
except ValueError:
    print("C")          # C
print("D")              # D
```

`A` prints before anything fails. `int("seven")` raises, so `B` never prints: like `return`, the line that fails ends the `try` block. With `int("7")`, the output would be `A`, `B`, `D`.

# Problem: Find the Bug

```
text = input("Enter a number: ")
try:
    value = flaot(text)
    print(value * 2)
except:
    print("Not a number.")
```

```
Enter a number: 3.5
Not a number.
```

`3.5` is a number. Why does every input print `Not a number.`?

# Solution: Find the Bug

```
text = input("Enter a number: ")
try:
    value = flaot(text)   # misspelled
    print(value * 2)
except ValueError:
    print("Not a number.")
```

```
NameError: name 'flaot' is not defined. Did you mean: 'float'?
```

`except:` with no name catches every exception, including the `NameError` from the typo, so the bug hides behind the wrong message. The textbook writes `except:` this way.

# Problem: Keep the Integers (us)

```
# Write keep_integers, a function that returns a new list of the
# entries that are integers, converted with int.
entries = ["12", "-3", "abc", "7", "4.5"]
print(keep_integers(entries))
```

```
[12, -3, 7]
```

- What should the `except` block do?
- Why not check with `isdigit` instead?

# Solution: Keep the Integers (us)

```
def keep_integers(entries):
    numbers = []
    for entry in entries:
        try:
            numbers.append(int(entry))
        except ValueError:
            pass
    return numbers

entries = ["12", "-3", "abc", "7", "4.5"]
print(keep_integers(entries))
```

`pass` is new; the next slide covers it. `isdigit` would reject `-3` (week 4), but `int` accepts it.

# New Keyword: `pass`

```
try:
    numbers.append(int(entry))
except ValueError:
    # skip bad entries

print(numbers)
```

```
try:
    numbers.append(int(entry))
except ValueError:
    pass

print(numbers)
```

```
IndentationError: expected an indented block after 'except' statement on line 3
```

Python refuses to run the left version. Every block needs at least one indented line, and a comment does not count (Indentation Errors). `pass` is a line that does nothing, so it fills a block that has nothing to do (right). It works in any block: `if`, `for`, `while`, `def`, or `except`.

# Parsons Problem #1 (you)

**One line does not belong.**

```
quantity = mystery("Quantity: ")
except ValueError:
return None
def mystery(prompt):
print(f"Total: ${price * quantity:.2f}")
try:
print("Enter a number.")
price = mystery("Price: ")
while True:
return float(input(prompt))
```

# Solution: Parsons Problem #1 (you)

```
def mystery(prompt):
    while True:
        try:
            return float(input(prompt))
        except ValueError:
            print("Enter a number.")

price = mystery("Price: ")
quantity = mystery("Quantity: ")
print(f"Total: ${price * quantity:.2f}")
```

The extra line was `return None`. `mystery` has to ask again, and `return None` would end the function on the first bad entry, and `price * quantity` would raise a `TypeError`. `return` ends the function, and the loop with it, the first time `float` succeeds. A bad entry prints the message, and `while True` asks again.

# Reading Files

# Files for Today

Download data.zip and unzip it next to your program. It makes the `data` folder on the right.

```
"data/names.txt"      # Mac
"data\\names.txt"     # Windows
```

The slides use the Mac path, which Windows also accepts; the cheat sheet explains why Windows has two. The next lecture has Python pick the separator.

```
week7/              ← open in VS Code
├── program.py
└── data/
    ├── measures.txt
    ├── months/
    ├── names.txt
    ├── prices.csv
    ├── products.csv
    ├── sales.csv
    └── scores.csv
```

## Opening a File with `with`

```
with open([FILENAME], "r") as [NAME]:
    # NAME is the open file; "r" means read

# the file is closed here
```

- `open` finds the file named `FILENAME` and opens it. The `"r"` is short for "read", so the line above it read a file.
- `with` keeps the file open for the indented block, then closes it, even if the block raises an exception.

## Without `with`: `open` and `close`

```
[NAME] = open([FILENAME], "r")
# NAME is the open file

[NAME].close()   # the file is closed here
```

Without `with`, you have to write a program like above. `close` is like clicking Save: until it runs, what the program wrote may not be in the file yet. `with` calls `close` for you.

## New Keyword: `finally`

```
try:
    # code that might raise an exception
except [EXCEPTION]:
    # runs only if the code in try raised EXCEPTION
finally:
    # always runs, whether or not the code in try raised
```

`finally` runs last, whether `try` finished or raised. The `except` is optional: a `try` can have only a `finally`.

## What `with` Does Underneath

```
with open([FILENAME], "r") as [NAME]:
    # NAME is the open file
```

```
[NAME] = open([FILENAME], "r")
try:
    # NAME is the open file
finally:
    [NAME].close()
```

Both do the same thing: `close` sits in `finally`, so it runs even if the block raises. Python added `with` in 2006 to replace this pattern.

# Problem: What Is the Output?

```
with open("data/names.txt", "r") as f:
    for line in f:
        print(line)
```

```
Ada
Grace
Alan
```

Right: `names.txt`, as it looks in VS Code.

## Solution: What Is the Output?

```
Ada

Grace

Alan
```

`for line in f:` hands the loop one line at a time, as a string. Each line keeps the `\n` that ends it in the file, so the first `line` is `"Ada\n"`, and `len(line)` is 4. `print` adds its own `\n`, so a blank line follows each name.

## Removing the `\n`

```
with open("data/names.txt", "r") as f:
    for line in f:
        print(line.strip())
```

```
Ada
Grace
Alan
```

`strip` removes spaces and newlines from both ends.

## `FileNotFoundError`

```
filename = "data/na.txt"
try:
    with open(filename, "r") as f:
        for line in f:
            print(line.strip())
except FileNotFoundError:
    print(f"Could not find {filename}")
```

```
Could not find data/na.txt
```

`open` raises a `FileNotFoundError` when no file has that name, such as `na.txt` typed for `names.txt`.

# Problem: What Is the Output?

```
total = 0
with open("data/measures.txt", "r") as f:
    for line in f:
        total += float(line)

print(f"Total: {total:.1f}")
```

`measures.txt` holds `10.1`, `9.9`, and `9.8`, one per line, then a blank last line.

## Solution: What Is the Output?

```
ValueError: could not convert string to float: '\n'
```

```
    for line in f:
        if line.strip() != "":
            total += float(line)
```

`float("10.1\n")` works, since `float` ignores spaces and newlines around the number (Converting Text to Numbers). The last line is only `"\n"`, with no number to convert. Skip blank lines before converting, and the program prints `Total: 29.8`.

# Lines with Fields

```
line = "Ada,90\n"
fields = line.strip().split(",")
print(fields)              # ['Ada', '90']

name = fields[0]
score = int(fields[1])
print(name, score + 5)   # Ada 95
```

Each line of `scores.csv` holds two **fields**, separated by a comma. Strip, split, then convert.

## Problem: What Is the Output?

```
with open("data/scores.csv", "r") as f:
    for line in f:
        fields = line.strip().split(",")
        print(fields[0], int(fields[1]))
```

## Solution: What Is the Output?

```
ValueError: invalid literal for int() with base 10: 'score'
```

The way to fix this is to read the header first, and then process the rest of the file:

```
with open("data/scores.csv", "r") as f:
    f.readline()
    for line in f:
        fields = line.strip().split(",")
        print(fields[0], int(fields[1]))
```

Nothing stores what `readline` gave back, so the header is thrown away.

# Problem: Total Sales (us)

```
# sales.csv holds one sale per line. Print the total of
# price * quantity over every sale, to the cent.
```

```
product,date,price,quantity
P-01,2023-01-01,9.95,5
P-02,2023-01-02,10.50,2
P-01,2023-01-03,9.85,4
P-02,2023-01-04,10.60,3
P-01,2023-01-05,9.95,1
```

```
Total: $151.90
```

- Which line is different from the rest?
- Which fields need converting, and to what?

# Solution: Total Sales (us)

```
total = 0
with open("data/sales.csv", "r") as f:
    f.readline()
    for line in f:
        fields = line.strip().split(",")
        price = float(fields[2])
        quantity = int(fields[3])
        total += price * quantity

print(f"Total: ${total:.2f}")
```

The header is the line that differs, so `readline` skips it. Price is field 2 and quantity is field 3. The product and the date are never used, so they stay strings. Quantity counts whole items, so it gets `int`.

# Parsons Problem #2 (you)

**One line does not belong.**

```
best_score = score
print(f"Top student: {best_name} ({best_score})")
f.readline()
if fields[1] > best_score:
score = int(fields[1])
best_name = ""
for line in f:
if score > best_score:
with open("data/scores.csv", "r") as f:
best_name = fields[0]
fields = line.strip().split(",")
best_score = -1
```

# Solution: Parsons Problem #2 (you)

```
best_name = ""
best_score = -1
with open("data/scores.csv", "r") as f:
    f.readline()
    for line in f:
        fields = line.strip().split(",")
        score = int(fields[1])
        if score > best_score:
            best_score = score
            best_name = fields[0]

print(f"Top student: {best_name} ({best_score})")
```

The extra line was `if fields[1] > best_score:`. `fields[1]` is a string like `"90"`, and comparing a string with a number raises a `TypeError`. `best_score` starts at `-1`, below any real score, so the first student always replaces it.

# Writing Files

## Opening a File for Writing

```
with open([FILENAME], "w") as [NAME]:
    # NAME is the open file; "w" means write

# the file is closed here
```

The only change from reading is `"w"`. If the file already exists, `"w"` **erases it** first.

# Writing with `write`

```
name = "Ada"
score = 90
with open("report.txt", "w") as f_out:
    f_out.write(f"Report for {name}\n")
    f_out.write("name,score\n")
    f_out.write(f"{name},{score}\n")
```

```
Report for Ada
name,score
Ada,90
```

This is `report.txt` after the program runs.

`write` puts the text in the file exactly as given, so end each line with `\n`.

`print(..., file=f_out)` works as well.

# Problem: Raise the Prices (us)

```
# Copy prices.csv (left) to new_prices.csv (right) with
# every price raised 10%, to the cent. Skip rows whose price is not
# a number, and print how many.
```

```
item,price
coffee,3.00
bagel,2.50
tea,n/a
muffin,2.80
juice,4.00
```

```
item,price
coffee,3.30
bagel,2.75
muffin,3.08
juice,4.40
```

- How do we copy the header?
- What does `float("n/a")` do?

# Solution: Raise the Prices (us)

```
skipped = 0
with open("data/prices.csv", "r") as f_in:
    with open("new_prices.csv", "w") as f_out:
        header = f_in.readline()
        f_out.write(header)
        for line in f_in:
            fields = line.strip().split(",")
            try:
                price = float(fields[1]) * 1.10
                f_out.write(f"{fields[0]},{price:.2f}\n")
            except ValueError:
                skipped += 1

print(f"Rows skipped: {skipped}")
```

Both files stay open at once, so the `with` blocks nest. The header keeps its `\n`, so `write` copies it as is. `float("n/a")` raises a `ValueError`, so the `write` after it never runs, the `tea` row is left out, and the program prints `Rows skipped: 1`.

# Opening a File for Appending

```
with open([FILENAME], "a") as [NAME]:
    # NAME is the open file; "a" means append

# the file is closed here
```

`"a"` keeps what is already in the file and adds to its end, like `append` on a list.

# Parsons Problem #3 (you)

**One line does not belong.**

```
f_out.write(f"{name}\n")
names = []
else:
with open("data/names.txt", "w") as f_out:
name = input("Name: ")
for line in f_in:
print(f"{name} is already on the list.")
with open("data/names.txt", "a") as f_out:
names.append(line.strip())
if name in names:
with open("data/names.txt", "r") as f_in:
```

# Solution: Parsons Problem #3 (you)

```
names = []
with open("data/names.txt", "r") as f_in:
    for line in f_in:
        names.append(line.strip())

name = input("Name: ")
if name in names:
    print(f"{name} is already on the list.")
else:
    with open("data/names.txt", "a") as f_out:
        f_out.write(f"{name}\n")
```

The extra line was `with open("data/names.txt", "w") as f_out:`. `"w"` would erase the list and leave only the new name.

# Summary

- An **exception** stops the program; `try` and `except` catch one, so the program can keep going
- Name the exception you expect, like `except ValueError:`; a bare `except:` also hides bugs such as typos
- `int` and `float` raise a `ValueError` on text that is not a number
- `with open(filename, "r") as f:` works like `try` with `f.close()` in `finally`, which always runs
- `for line in f:` reads one line at a time, as a string that keeps its `\n`, so `strip` it
- `f.readline()` reads one line (`""` at the end of the file), which is how to skip a header
- `split(",")` breaks a line into fields; convert number fields with `int` or `float`
- `open(filename, "w")` creates or erases a file, and `"a"` adds to its end; `f_out.write(text)` adds `text` with no `\n` of its own

## Next Up: The `os` Module

The `os` Module

The Exceptions and Files Cheat Sheet has today's tools in one place, with more examples and the textbook's `readline` loop.