Finger Exercises 16

Assertions & Exceptions

≈ 60 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class. Most of the code here is new, so you have to *work it out*, not remember it. For every line, ask: *can this line fail? If it fails, which* `except` *catches it — or does the program crash?*
How to do it. On **paper**, **no computer**, in one sitting. If an item takes you more than a minute, skip it and come back. Parts A–M are for *everyone*; Part N is a **stretch**. The Console Quests are for tonight, *at* a computer. The answer key is on the last pages — do not peek until you are done.
Predict first, then check. Write down what you think *before* you look anything up or type anything in — even for the Console Quests. A guess that turns out wrong teaches you far more than an answer you only watched the computer produce.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Name the exception each line raises. (`L` has 3 items.)

`L[100]` _____ `int('abc')` _____

`10 / 0` _____ `'a' / 4` _____

A2. An assertion is written `assert` _____`, "message"`. If the condition is **False** it raises _____ and the program _____.

A3. In a `try` statement, the `else` block runs only when the `try` block _____. The `finally` block runs _____. An `except ValueError:` block catches _____ and nothing else.

A4. Circle **True** or **False**:

(a) A bare `except:` catches every kind of error. True / False

(b) `finally` is skipped when the program crashes. True / False

(c) Assertions are for catching the programmer's own mistakes. True / False

(d) A printed warning is easier to ignore than a raised exception. True / False

(e) `except ValueError:` also catches a `ZeroDivisionError`. True / False

(f) An `assert` whose condition is `True` prints nothing, and the program goes on. True / False

(g) Only Python can raise an exception; your own code cannot. True / False

(h) After a `raise`, the rest of the function still runs. True / False

Part B • Name the Error

CLASSIFY ≈ 5 min

Use `L = [10, 20, 30]`, `t = (1, 2)` and `s = 'cat'`. No variable called `total` exists. For each line, write the name of the exception it raises. If there is **no error**, write the value instead.

Line	Error or value	Line	Error or value
`L[3]`		`L[-3]`	
`int('42')`		`int('4.2')`	
`int(4.2)`		`'7' + 3`	
`'7' * 3`		`7 / 0`	
`0 / 7`		`7 % 0`	
`s[3]`		`t[0] = 5`	
`print(total)`		`int('')`	

Part C • What Does It Print?

TRACE ≈ 6 min

Write every line of output, or the value returned.

C1.

```
def sum_digits(s):
    total = 0
    for ch in s:
        try:
            total += int(ch)
        except:
            pass
    return total
```

`sum_digits('12E4')`
returns:

_____

C2. The `try` has moved: it now holds the whole loop.

```
def total_digits(s):
    total = 0
    try:
        for ch in s:
            total += int(ch)
    except ValueError:
        pass
    return total
```

`total_digits('12E4')`
returns:

_____

`total_digits('E124')`
returns:

_____

C3. One word changed between these two. Write both outputs.

```
try:
    n = int('abc')
    print(100 / n)
except ValueError:
    print('Bad input!')
else:
    print('Done!')
```

Output: _____

```
try:
    n = int('abc')
    print(100 / n)
except ValueError:
    print('Bad input!')
finally:
    print('Done!')
```

Output: _____

C4.

```
bad = 0
for w in ['4', 'x', '10', '', '3.0']:
    try:
        n = int(w)
    except ValueError:
        bad += 1
print(bad)
```

Output:

_____

C5.

```
def f(x):
    try:
        return 10 / x
    finally:
        print('checked')

print(f(2))
```

Output (every line, in order):

_____

_____

Part D • Crash or No Crash?

TRACE ≈ 3 min

```
1 def safe_divide(a, b):
2     try:
3         return a / b
4     except ZeroDivisionError:
5         return None
```

Write the value returned, or the exception that crashes the program.

D1. `safe_divide(10, 2)` _____

D2. `safe_divide(10, 0)` _____

D3. `safe_divide(10, 'a')` _____

D4. `safe_divide(0, 10)` _____

D5. `safe_divide('10', 2)` _____

Part E • One Program, Five Inputs

TRACE ≈ 6 min

```
1 try:
2     n = int(input())
3     r = 100 / n
4     print('A')
5 except ValueError:
6     print('B')
7 except ZeroDivisionError:
8     print('C')
9 else:
10    print(r)
11 finally:
12    print('E')
13 print('F')
```

The user types the input, then presses Enter. Write every line the program prints, top to bottom, separated by slashes (`A / 20.0 / ...`).

Input	Output
`5`	
`abc`	
`0`	
`2.5`	
`-4`	

E1. Now delete lines 7 and 8 (the `ZeroDivisionError` branch) and run it again with input `0`. Write everything that appears, in order, and say what happens to `F`.

_____

Part F • Which Lines Run?

TRACE ≈ 5 min

```
1  text = input()
2  try:
3      n = int(text)
4      half = 10 / n
5  except ValueError:
6      half = -1
7  else:
8      print('ok')
9  finally:
10     print('end')
11 print(half)
```

For each input, tick every line that **runs**. A line that starts and then raises an error *does* count as run.

Line		4	x	0
3	`n = int(text)`	☐	☐	☐
4	`half = 10 / n`	☐	☐	☐
6	`half = -1`	☐	☐	☐
8	`print('ok')`	☐	☐	☐
10	`print('end')`	☐	☐	☐
11	`print(half)`	☐	☐	☐

F1. Write what the program prints for each input.

4: _____ x: _____ 0: _____

Part G • The User Types. . .

TRACE ≈ 5 min

The user types the lines shown, one each time the program asks.

G1.

```
1 while True:
2     try:
3         n = int(input('Enter an integer: '))
4         break
5     except:
6         print('Not an integer! Try again.')
7 print('Got', n)
```

Typed: `seven`, then `7.0`, then nothing (just Enter), then `7`, then `8`.

(a) How many prompts are shown? _____

(b) How many messages are printed? _____

(c) The last line printed: _____

(d) What happens to the 8? _____

G2.

```
1  while True:
2      try:
3          n = int(input('Age: '))
4      except ValueError:
5          print('Numbers only!')
6      else:
7          if n > 0:
8              break
9          print('Too small!')
10 print('Age', n)
```

Typed: `abc`, `-5`, `0`, `19`.

Write every line the program prints (not the prompts).

_____

_____

_____

_____

Part H • You Raise It, the Caller Catches It

TRACE ≈ 5 min

H1.

```
def check_age(age):
    if age < 0:
        raise ValueError('negative age')
    print('ok', age)
    return age

for a in [20, -3, 15]:
    try:
        check_age(a)
        print('saved')
    except ValueError:
        print('skipped')
```

Write every line of output.

_____

_____

_____

_____

_____

H2.

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError('Not enough funds')
    return balance - amount

balance = 100
for amount in [30, 90, 50]:
    try:
        balance = withdraw(balance, amount)
    except ValueError:
        print('Refused', amount)
print(balance)
```

Write every line of output.

_____

_____

Why does a refused amount leave `balance` as it was?

_____

_____

H3. An uncaught error ends with one line: its name, a colon, and its message, like this: `ValueError: Not enough funds`

```
def sum_digits(s):
    total = 0
    for ch in s:
        if not ch.isdigit():
            raise ValueError(f"'{ch}' is not a digit!")
        total += int(ch)
    return total
```

For each call, write that last line, or the value returned if there is no error.

`sum_digits('12E4')` _____ `sum_digits('305')` _____

`sum_digits('9 1')` _____ `sum_digits('')` _____

Part I • Assertions

TRACE ≈ 5 min

I1.

```
1 def avg(grades):
2     assert len(grades) != 0, 'No grades!'
3     return sum(grades) / len(grades)
4
5 def divide(a, b):
6     assert b != 0, 'b cannot be 0'
7     return a / b
```

Write the value returned, or the last line of the error.

`avg([80, 90])` _____

`avg([0])` _____

`avg([])` _____

`divide(0, 5)` _____

`divide(5, 0)` _____

I2. Each line is typed into the Shell. Write what appears, or *nothing*.

`assert 2 + 2 == 4, 'maths works'` _____

`assert 2 + 2 == 5, 'maths is broken'` _____

`assert 2 + 2 == 5` _____

`assert 3 > 1 and 1 > 3, 'order'` _____

I3. A docstring says what a function expects. Write one `assert` line (with a message) for each promise.

(a) `name` is not an empty string. `assert` _____

(b) `L1` and `L2` have the same length. `assert` _____

(c) `mark` is from 0 to 100 (both allowed). `assert` _____

(d) `n` is an `int`. `assert` _____

Part J • Four Versions of avg

CLASSIFY ≈ 4 min

For each version and each call, write the value returned. Add anything printed, or the error that stops the program.

Version 1

```
def avg(grades):
    return sum(grades) / len(grades)
```

Version 2

```
def avg(grades):
    try:
        return sum(grades) / len(grades)
    except ZeroDivisionError:
        print('warning: no grades')
        return None
```

Version 3

```
def avg(grades):
    if len(grades) == 0:
        return 0.0
    return sum(grades) / len(grades)
```

Version 4

```
def avg(grades):
    assert len(grades) > 0, 'No grades!'
    return sum(grades) / len(grades)
```

	`avg([90, 70])`	`avg([])`
Version 1		
Version 2		
Version 3		
Version 4		

Part K • Which Tool?

CLASSIFY ≈ 3 min

Tick the tool the lecture's flowchart chooses for each job.

Job	try/except	raise	assert
The user typed `hello` instead of a number; ask again.	☐	☐	☐
`withdraw` is asked for more money than the balance.	☐	☐	☐
A student in the class list has no grades; print a warning and carry on.	☐	☐	☐
`avg` was handed an empty list, which should never happen.	☐	☐	☐
One line of a marks list says `absent`; skip it and add up the rest.	☐	☐	☐
`pairwise_div` is called with lists of different lengths; the caller must be told.	☐	☐	☐
Your own `count_vowels` must never return a negative number; check it before the `return`.	☐	☐	☐
`set_mark(m)` is called with `m = 150`.	☐	☐	☐

Part L • Circle the Bug, Then Fix It

FIX ≈ 8 min

L1. The user types `12`, yet the program still prints `Not a number!`

```
try:
    n = int(input('Number: '))
    print(nn * 2)
except:
    print('Not a number!')
```

Circle the line that hides the real problem. Fix: _____

L2. The apology message is never printed.

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError('Too much')
        print('Sorry, too much!')
    return balance - amount
```

Why never? _____

L3. The user types `abc` and the program crashes, even though there is a `try`.

```
try:
    n = int(input('Number: '))
    print(100 / n)
except ZeroDivisionError:
    print('Cannot divide by zero!')
```

Which error escapes? _____ Fix: _____

L4. The user types `abc`. The program prints `Not a number!` — and then crashes anyway.

```
try:
    n = int(input('Number: '))
except ValueError:
    print('Not a number!')
print(n * 2)
```

Name the second error: _____ Fix: _____

L5. The user types `5`, and the program asks again. And again. It never stops asking.

```
while True:
    try:
        n = int(input('Integer: '))
    except ValueError:
        print('Try again!')
print('Got', n)
```

What is missing, and where? _____

L6. `avg([80, 90])` halts with `AssertionError: No grades!`

```
def avg(grades):
    assert len(grades) == 0, 'No grades!'
    return sum(grades) / len(grades)
```

Fix line 2: _____

Part M • One Sentence Each

EXPLAIN ≈ 3 min

M1. Why is raising an exception better than printing a warning and returning the old balance?

_____

M2. In one sentence, how is an assertion different from an exception?

_____

M3. Why should the `try` block hold only the line that can fail, with the rest in `else`?

_____

Part N • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full. Run the sample call on paper before you call it done.

N1. `read_mark()`: keep asking with the prompt `'Mark: '` until the user types a whole number from 0 to 100. Print `Numbers only!` for text and `0 to 100!` for a number out of range. Return the mark.
If the user types `ten`, `150`, `75`, it prints the two messages and returns `75`.

N2. `first_number(words)`: return the first item of the list `words` that `int` accepts, as an `int`. If there is none, `raise ValueError('no numbers')`.
`first_number(['hi', '3.5', '42', '7'])` is 42.

Console Quests

tonight · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type.

1. Type the four `assert` lines of I2 into Thonny's Shell, one at a time. Did each one show what you wrote?
2. Save the program of Part F as a file and run it with the input `0`. Read the red error: which line number does it name? Is `end` printed above the error or below it?
3. Type `int(' 42 ')`, `int('4 2')` and `int('-7')`. One of them surprises most people. Then define `f` from C5 and call `f(0)`: does `checked` appear before the error?

Stuck on more than two items? Re-read the Lecture 16 slides "Common Exception Types", "`else`: Tracing", "`finally`: Tracing", "`print` vs `raise`" and "When to Use Each?" before the next class — then redo Parts E, F and H cold. For every line, ask: *can it fail, and who catches it?* If no `except` names the error, the program crashes — after `finally` has had its say.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. `IndexError`; `ValueError`; `ZeroDivisionError`; `TypeError`.

A2. `assert <condition>, "message"`; it raises `AssertionError` and the program **halts** immediately.

A3. finishes **with no error**; **always** (error or not, caught or not); only a `ValueError`.

A4. (a) **True** — and that is the danger: it also hides bugs you wanted to see. (b) **False** — `finally` always runs, even on the way out of a crash. (c) **True** — exceptions are for the user and the world. (d) **True** — an exception cannot be ignored. (e) **False** — a named `except` catches that one exception only. (f) **True** — a passing assert is silent. (g) **False** — `raise` lets your code raise one. (h) **False** — `raise` leaves the function on the spot, like `return`, but with an error.

B. Left column: `L[3]` **IndexError** (the indices are 0 to 2); `int('42')` **42**; `int(4.2)` **4** (a float is cut, not refused); `'7' * 3` **'777'**; `0 / 7` **0.0** (zero on *top* is fine); `s[3]` **IndexError**; `print(total)` **NameError**.
Right column: `L[-3]` **10**; `int('4.2')` **ValueError** (the *string* must look like a whole number); `'7' + 3` **TypeError**; `7 / 0` **ZeroDivisionError**; `7 % 0` **ZeroDivisionError** (`%` divides too); `t[0] = 5` **TypeError** (a tuple cannot be changed); `int('')` **ValueError**.

C1. `7` — not `10` and not a crash. `int('E')` raises `ValueError`, the `except` swallows it and `pass` skips that character, so only $1 + 2 + 4$ is added.

C2. `3`, then `0`. The error on 'E' jumps *out of the loop* to the `except`, so the loop never comes back for the '4'. With 'E' first, nothing was added yet. Where the `try` sits decides how much work an error throws away.

C3. Left prints only `Bad input!` — `else` runs *only* when the `try` block finished with no error. Right prints `Bad input!` then `Done!` — `finally` always runs.

C4. `3`: 'x', the empty string '' and '3.0' all fail. `int` takes a string only when it looks like a whole number.

C5. `checked`, then `5.0`. The `return` is on its way out, but `finally` still runs first, so its `print` comes before the caller's.

D1. `5.0` — no error, so the `try` block simply returns.

D2. `None` — `ZeroDivisionError` is caught by name and the function returns `None` instead of crashing.

D3. **Crash:** `TypeError`. The `try/except` does *not* protect you here — it names only `ZeroDivisionError`, and a `TypeError` is a different exception, so it flies straight past.

D4. `0.0` — zero on top is no problem.

D5. **Crash:** `TypeError` — '10' is a string, not a number. Same reason as D3.

E. `5`: `A / 20.0 / E / F`. `abc`: `B / E / F`. `0`: `C / E / F` (line 3 fails, so `A` is never printed). `2.5`: `B / E / F` — `int` refuses '2.5', so it is a `ValueError`. `-4`: `A / -25.0 / E / F`. `E` appears in every row; `r` is printed only when nothing went wrong.

E1. `E`, and then the program **crashes** with `ZeroDivisionError`. Nothing catches it, but `finally` still runs on the way out. `F` is **never printed**: the program is dead before it reaches `print('F')`.

F. Input `4`: lines 3, 4, 8, 10, 11. Input `x`: lines 3, 6, 10, 11 (line 3 fails, so line 4 is skipped). Input `0`: lines 3, 4, 10, then a crash — line 4 fails with a `ZeroDivisionError` that no `except` names, so line 11 never runs. Line 10 is ticked in every column: that is `finally`.

F1. `4`: `ok / end / 2.5`. `x`: `end / -1`. `0`: `end`, then the crash.

G1. (a) **4**. (b) **3** — `seven`, `7.0` and the empty line all fail. (c) `Got 7`. (d) It is **never read**: line 4's `break` ended the loop after `7`. Line 4 is reached only when line 3 did not fail.

G2. `Numbers only!` / `Too small!` / `Too small!` / `Age 19`. `-5` and `0` are good integers, so the `else` runs; they fail the *program's* test, not `int`.

H1. `ok 20` / `saved` / `skipped` / `ok 15` / `saved`. For `-3` the `raise` leaves `check_age` at line 3, so neither `ok -3` nor `saved` is printed. The function finds the problem; the caller decides what to do.

H2. `Refused 90` / `20`. After `30` the balance is `70`, so `90` is too much; `50` then leaves `20`. On line 9 the `withdraw` call fails *before* the `=` can store anything, so `balance` keeps its old value.

H3. '12E4': `ValueError: 'E' is not a digit!`; '305': `8`; '9 1': `ValueError: ' ' is not a digit!` (the space is a character too, and not a digit); '': `0` — the loop never runs, so nothing is checked and nothing raised.

I1. `85.0`; `0.0` (one grade of zero is still a grade: the length is 1); `AssertionError: No grades!`; `0.0`; `AssertionError: b cannot be 0`. The assert stops the call *before* the division, so no `ZeroDivisionError` is ever reached.

I2. *nothing*; `AssertionError: maths is broken`; `AssertionError` alone (no message was given); `AssertionError: order` — the whole condition is `False`, because its second half is.

I3. (a) `assert len(name) != 0, 'name is empty'` (or `name != ''`). (b) `assert len(L1) == len(L2), 'different lengths'`. (c) `assert 0 <= mark <= 100, 'bad mark'` (or `mark >= 0 and mark <= 100`). (d) `assert type(n) == int, 'must be an integer'`. Each condition says what must be **true**; the message says what went wrong.

J. `avg([90, 70])` is `80.0` for **all four**: the versions differ only when something goes wrong. `avg([])`: Version 1 **crashes** with `ZeroDivisionError`; Version 2 prints `warning: no grades` and returns `None`; Version 3 returns `0.0` (the guard clause); Version 4 **halts** with `AssertionError: No grades!`. Which is best depends on what an empty list *means* in your program.

K. **try/except, raise, try/except, assert, try/except, raise, assert, raise.** Rows 1, 3 and 5 can recover and keep going. Rows 2, 6 and 8 are bad data from the *caller*, so `raise` and let the caller decide. Rows 4 and 7 say "should never happen" about our own code — a bug, which is exactly what `assert` is for.

L1. The bare `except:` on line 4 is the culprit — the typo `nn` raises `NameError`, which the bare `except` swallows and then reports as bad input. Write `except ValueError:` so only the error you meant to handle is caught (and the typo shows itself).

L2. `raise` stops the function on the spot, so any line written after it can never run. Put the `print` *before* the `raise`, or delete it and let the message inside `ValueError` do the talking.

L3. `ValueError` escapes: the only `except` names `ZeroDivisionError`. Add a second branch, `except ValueError:` with `print('Not a number!')`.

L4. `NameError`: `int` failed, so `n` was never made, and line 5 still runs. Indent line 5 under an `else:` so it runs only when the `try` worked.

L5. A `break`. Put it after line 3, inside the `try` (or under an `else:`), so the loop ends once `int` has worked. Line 6 is never reached as written.

L6. The condition is the wrong way round. An assert states what must be **true**: `assert len(grades) != 0, 'No grades!'` (or `> 0`).

M1. A print is easy to miss and the caller silently receives wrong data, while an exception cannot be ignored and stops the bad data at its source.

M2. An assertion checks something the programmer believes must be true and halts if it is not; an exception handles a runtime problem the program may be able to recover from.

M3. So that the `except` catches only the error you expected from that line, not a different error (or a bug) in code that was never meant to fail.

N1. `while True:` / `try:` / `m = int(input('Mark: '))` / `except ValueError:` / `print('Numbers only!')` / `else:` / `if 0 <= m <= 100:` / `return m` / `print('0 to 100!')`. A `return` inside the loop ends it as well as `break` does. This is G2 with a second limit.

N2. `for w in words:` / `try:` / `return int(w)` / `except ValueError:` / `pass` / then, after the loop, `raise ValueError('no numbers')`. The `raise` must stand *outside* the loop: one word that fails says nothing about the words after it.