Programming with Python

Stefan Güttel, guettel.com (http://guettel.com)

Contents:

1. Control flow
2. Finer control of the loops
3. Breaking a function execution
4. Exceptions
5. Doing nothing
6. Breaking out of several loops

Control flow

To quote from Wikipedia's definition of control flow (http://en.wikipedia.org/wiki/Control_flow):

> In computer science, control flow (or alternatively, flow of control) refers to the order in which the individual statements, instructions or function calls of an imperative or a declarative program are executed or evaluated.

We have already seen some methods of controling the flow:

- loops: *repeat this part of the code as long as some condition is met*,
- branches: *execute different parts of the code, depending on some condition(s)*,
- functions: *execute a part of the code that is located somewhere away from the currently executed line (i.e., not immediately after it).*

However, there are various ways to affect the flow even further, thus achieving a finer control of the execution, a simpler or a more readable code, or a better error control.

Finer control of loops

We have already seen loops in Python:

```
for some_variable in range(arguments):
    do_something
```

or

```
while some_condition:
    do_something
```

Recall that the body of the loop is executed completely, i.e., **the loop's condition gets checked only before the body's execution, not during.**

So, how can we handle the following two cases **in the middle of the body** (i.e., not in the beginning or the end)?

1. *If some condition is met, exit the loop.*
2. *If some condition is met, stop executing the body and go back to the begining of the loop.*

`break` and `continue`

Example: Write a program that loads integers until a zero is loaded. The program has to write the sum of all the digits of all the positive loaded numbers (ignoring the negative ones).

Let us first define the function to compute the sum of the digits of a **positive** integer (to reduce the amount of typing later, as well as to make the code more readable):

In [1]:

```
def digitsum(n):
    """
    Return the sum of the digits of `n`, which is assumed to be a positive integer.
    """
    s = 0
    while n > 0:
        s += n % 10
        n //= 10
    return s
```

Let us now solve the problem using only what we've learned so far:

In [2]:

```
s = 0
last = False
while not last:
    n = int(input("n = "))
    if n == 0:
        last = True
    elif n > 0:
        s += digitsum(n)
print("The sum of the digits of positive integers:", s)
```

```
n = 17
n = -17
n = 0
The sum of the digits of positive integers: 8
```

Alternatively, we could have done this:

In [3]:

```
s = 0
n = int(input("n = "))
while n != 0:
    if n > 0:
        s += digitsum(n)
    n = int(input("n = "))
print("The sum of the digits of positive integers:", s)
```

```
n = 17
n = -17
n = 0
The sum of the digits of positive integers: 8
```

Notice that each of these simple examples is clumsy:

- The first solution uses an extra variable `last` and has most of the body wrapped in an `if` or `else` block.
- The second solution also has most of the body wrapped in an `if` or `else` block.
  Further, the body is badly organized: it first handles some `n` loaded before and then loads a new version of `n` (which naturally belongs to the next run of the body).

These may not seem like big issues, but do remember that this is just a simple example. Literally every part of the above code can be more complex:

- `input` could have been some complex data gathering (i.e., from the internet). Notice that, in the second case, we would either need to copy such a more complex loading twice in the code, or create a separate function (which could be messy if there is a big number of arguments to pass to such a function).
- The conditions `n != 0` and `n > 0` could have been far more complex (for example, some data analysis).
- We may have encountered far more than just two simple conditions, which could have easily lead to very deep indentation (`if` inside `if` inside `if` inside...).

So how to do it better?

A better way

Let us write the algorithm in English:

> Load numbers until you load a zero.
> Ignore each negative loaded number `n`.
> For positive `n` compute the sum of its digits and add it to the global sum. Print the solution.

Let us now rewrite it a bit (but still in English), to better resemble a program structure:

> Repeat this:
>
> > Load `n`.
> > If `n == 0`, **break** this loop.
> > If `n < 0`, **continue** this loop (i.e., skip the rest of the body and go back to loading a number).
> > Increase `s` by the sum of digits of `n`.
>
> Print `s`.

Python has statements `break` and `continue` that behave exactly the way our "English code" expects them to:

- `break` -- break the loop (i.e., continue the execution on the first command **after** the loop's body),
- `continue` -- stop the current execution of the body, but don't exit the loop (i.e., continue the execution on the **loop's header**).

The only question is how to handle "repeat this". Notice that this is equivalent to "repeat this unconditionally", which in Python is written as

```
while True:
    ...
```

The Python code

We are now ready to write the program neatly:

In [4]:

```
s = 0
while True:
    n = int(input("n = "))
    if n == 0:
        break
    if n < 0:
        continue
    s += digitsum(n)
print("The sum of the digits of positive integers:", s)
```

```
n = 17
n = -17
n = 0
The sum of the digits of positive integers: 8
```

Observe how the code is now naturally organized and easy to read:

1. `while True`, a so called *infinite loop*, tells us that the execution will be terminated somewhere in the body, most likely depending on the input. This means that the final input will **probably** be ignored. Otherwise, we'd break the loop after the body's execution, i.e., using an appropriate loop condition.
2. The first thing we do in the body is load the data that will be processed in that execution of the body.
3. We have some condition that leads to `break`ing, which means that this is the condition to stop.
   If we had more than one such condition, we could have easily written

   ```
   if condition1:
       break
   if condition2:
       break
   ...
   ```

   which would be a very easy to read sequance of obviously breaking conditions.

Remarks

- `break` and `continue` exist in most of the modern languages (including Python 2) and have the same functionality.
- "Infinite" loops are not directly related to either `break` or `continue`.
  However, remember the (somewhat informal) definition of an algorithm:

  > A **sequence** of actions that are always executed in a **finite** number of steps, used to solve a certain problem.

  It is very important to **always terminate your loops**, be it via their conditions or a `break` statement or some other breaking technique.
- The `break` and `continue` affect the innermost loop. In other words, in the following code

  ```
  while condition1:
      ...
      while condition2:
          ...
          if condition3:
              break
          ... # this is skipped by break
      ... # break "jumps" here
  ... # not here
  ```

  the `break` statement will break only the inner `while` loop (the one with a condition `condition2`).
- The `break` and `continue` have no meaning outside of the loops (for example, in an `if` block that is not inside of some loop). Using them there will cause a syntax error (*SyntaxError: 'break' outside loop*) as soon as you try to run your program. Even this will not work:

In [5]:

```
def f():
    break

for i in range(17):
    f()
```

```
  File "<ipython-input-5-248c41f8ba9d>", line 2
    break
    ^
SyntaxError: 'break' outside loop
```

Loading a list

We had this example two weeks ago:

> **Example:** Input integers until -17 is loaded, then print them all (**except** -17) sorted descendingly.

This was the solution:

In [6]:

```
lst = list()  # Create an empty list
not_done = True
while not_done:
    x = int(input("Please type an integer: "))
    if x == -17:
        not_done = False
    else:
        lst.append(x)
print("The sorted list:", sorted(lst, reverse=True))
```

```
Please type an integer: 17
Please type an integer: 19
Please type an integer: 13
Please type an integer: -17
The sorted list: [19, 17, 13]
```

However, using a `break` is much more natural here:

In [7]:

```
lst = list()  # Create an empty list
while True:
    x = int(input("Please type an integer: "))
    if x == -17:
        break
    lst.append(x)
print("The sorted list:", sorted(lst, reverse=True))
```

```
Please type an integer: 17
Please type an integer: 19
Please type an integer: 13
Please type an integer: -17
The sorted list: [19, 17, 13]
```

Similarly, if we want to **keep** -17 in the list, we just append the new number **before** checking if the loop should stop:

In [8]:

```
lst = list()  # Create an empty list
while True:
    x = int(input("Please type an integer: "))
    lst.append(x)
    if x == -17:
        break
print("The sorted list:", sorted(lst, reverse=True))
```

```
Please type an integer: 17
Please type an integer: 19
Please type an integer: 13
Please type an integer: -17
The sorted list: [19, 17, 13, -17]
```

Using `else` with loops

As we have mentioned before, it is possible to use `else` in conjuction with loops in Python.

The body of a loop's `else` is executed if the loop was **not terminated by a `break` statement** (but by its condition).

Observe the following example (try running it twice: once with loading a zero, and once with loading a negative number):

In [9]:

```
n = 17
while n > 0:
    n = int(input("n = "))
    if n == 0:
        break
else:
    print("The loop was terminated with a negative number (not a zero!).")
```

```
n = -17
The loop was terminated with a negative number (not a zero!).
```

Warning: If you use this, be careful with your indentation! The above is **not** the same as

In [10]:

```
n = 17
while n > 0:
    n = int(input("n = "))
    if n == 0:
        break
    else:
        print("The loop was terminated with a negative number (not a zero!).")
```

```
n = 17
The loop was terminated with a negative number (not a zero!).
n = 19
The loop was terminated with a negative number (not a zero!).
n = -17
The loop was terminated with a negative number (not a zero!).
```

Breaking a function execution

This one we already know, but it is worth mentioning it here as well:

> `return` returns a value from the function, but it also **immediatelly terminates the function's execution**.

This makes it very convenient to do various tasks without using some extra variables. Let us observe an example:

In [11]:

```
n = int(input("n = "))
if n < 2:
    is_prime = False
else:
    is_prime = True
if is_prime:
    for d in range(2, n):
        if n % d == 0:
            is_prime = False
            break
if is_prime:
    print("The number", n, "is a prime.")
else:
    print("The number", n, "is not a prime.")
```

```
n = 26
The number 26 is not a prime.
```

A somewhat shorter (albeit a bit more cryptic) version:

In [12]:

```
n = int(input("n = "))
is_prime = (n >= 2)
if is_prime:
    for d in range(2, n):
        if n % d == 0:
            is_prime = False
            break
if is_prime:
    print("The number", n, "is a prime.")
else:
    print("The number", n, "is not a prime.")
```

```
n = 26
The number 26 is not a prime.
```

Let us now observe how this can be done with a function:

In [13]:

```
def is_prime(n):
    """
    Return True if `n` is a prime; False otherwise.
    """

    if n < 2: return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True

n = int(input("Input an integer: "))
if is_prime(n):
    print("The number", n, "is a prime.")
else:
    print("The number", n, "is not a prime.")
```

```
Input an integer: 26
The number 26 is not a prime.
```

Observe the natural organization of the function code:

1. If `n < 2`, we know that `n` is not a prime, so **stop analyzing it** and just return `False`.
2. Check the potential divisors $(2, 3, \ldots, n-1)$. If any of them divides `n`, we have found out that `n` is not a prime, so **stop analyzing it** and just return `False`.
3. After the loop, if we are still here, that means that we found no divisors (otherwise, the `return` in the loop would stop the function's execution), which means that the number is a prime, so return `True`.

Note: A more Pythonic way to do the above would be:

In [14]:

```
def is_prime(n):
    """
    Return True if `n` is a prime; False otherwise.
    """
    return (n > 1) and all(n % d > 0 for d in range(2, n))

n = int(input("Input an integer: "))
if is_prime(n):
    print("The number", n, "is a prime.")
else:
    print("The number", n, "is not a prime.")
```

```
Input an integer: 26
The number 26 is not a prime.
```

Exceptions

We have seen how to break loops and how to break functions. In those terms, exceptions could be regarded as *break anything (no matter how deep)*.

The basic use of exceptions, written in plain English, is like this:

> Do this:
>
> > Some work (may include loops, function calls, more exceptions,... anything!)
>
> If something "unexpected" happened,
>
> > Handle it like this.

The simpliest example of an exception is a division by zero. See what happens if you try to do it in the following simple program that loads `a` and `b` and computes $\sqrt{a/b}$.

In [15]:

```
from math import sqrt
a = float(input("a = "))
b = float(input("b = "))
print("sqrt({} / {}) = {}".format(a, b, sqrt(a/b)))
```

```
a = 1
b = 0

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-15-d4bcbb23fd21> in <module>()
      2 a = float(input("a = "))
      3 b = float(input("b = "))
----> 4 print("sqrt({} / {}) = {}".format(a, b, sqrt(a/b)))

ZeroDivisionError: float division by zero
```

Notice the last line?

```
ZeroDivisionError: float division by zero
```

This is an unhandled exception. What happened is that `a/b` produced an error which then exits the current block of execution (be it `if`, `for`, a function,...), then the one that's surrounding it, and so on, until it is either handled or it exits the program.

Since we didn't handle that exception, the program was terminated, with a rather ugly (but informative!) message.

See what happens when we handle an exception:

In [16]:

```
from math import sqrt
try:
    a = float(input("a = "))
    b = float(input("b = "))
    print("sqrt({} / {}) = {}".format(a, b, sqrt(a/b)))
except Exception:
    print("Your a and/or b are wrong.")
```

```
a = 1
b = 0
Your a and/or b are wrong.
```

It is possible to make a general `exception` block that catches all the exceptions:

```
try:
    ...
except:
    ...
```

However, **never do that**, as it masks the errors you did not anticipate and disables breaking the program with `Ctrl+C` and similar mechanisms.

At minimum, use `except Exception`. However, it is much better to catch more specific exception(s) when possible.

Distinguishing between different exceptions

Notice how both `(a,b) = (1,0)` and `(a,b) = (-1,1)` will give you the same error (that *your a and/or b are wrong*), even though these are different errors: the first input causes the divison by zero, while the second one contains the real square root of a negative number. So, how do we distinguish between these two?

Notice the `ZeroDivisionError` above? This is the name of the exception that has happened. We can specify it, so that our `try...except` block handles exactly this error (while leaving all other errors unhandled):

In [17]:

```
from math import sqrt
try:
    a = float(input("a = "))
    b = float(input("b = "))
    print("sqrt({} / {}) = {}".format(a, b, sqrt(a/b)))
except ZeroDivisionError:
    print("Division by zero.")
```

```
a = 1
b = 0
Division by zero.
```

What happens if you load `(a,b) = (-1,1)` in the above code?

To handle more than one exception, we can just add more `except` blocks (using the names of the appropriate exceptions):

In [18]:

```
from math import sqrt
try:
    a = float(input("a = "))
    b = float(input("b = "))
    print("sqrt({} / {}) = {}".format(a, b, sqrt(a/b)))
except ZeroDivisionError:
    print("Division by zero.")
except ValueError:
    print("a/b does not have a real square root!")
```

```
a = -1
b = 1
a/b does not have a real square root!
```

Remark: complex arithmetic in Python

Python can do complex arithmetic:

In [19]:

```
from cmath import sqrt
try:
    a = float(input("a = "))
    b = float(input("b = "))
    result = sqrt(a/b)
    if result.imag > 0:
        print("sqrt({} / {}) = {}i".format(a, b, result.imag))
    else:
        print("sqrt({} / {}) = {}".format(a, b, result.real))
except ZeroDivisionError:
    print("Division by zero.")
except ValueError:
    print("a/b does not have a square root!")
```

```
a = -1
b = 1
sqrt(-1.0 / 1.0) = 1.0i
```

Note the following two details:

1. The imaginary unit in Python is displayed as `j` (instead of `i`), which is the reason why we handled the two cases ourselves instead of directly printing the result.
2. The `except ValueError` block in the above code is useless, as there is no input that can cause a `ValueError` with a complex square root.

`else` = "all is fine"

A `try...except` block can also have an `else` part. This one is executed if no exception has happened in the `try` block. For example:

In [20]:

```
from math import sqrt
try:
    a = float(input("a = "))
    b = float(input("b = "))
    result = sqrt(a/b)
except ZeroDivisionError:
    print("Division by zero.")
except ValueError:
    print("a/b does not have a square root!")
else:
    print("sqrt({} / {}) = {}".format(a, b, result))
```

```
a = 17
b = 19
sqrt(17.0 / 19.0) = 0.9459053029269173
```

This is very similar to the previuous code (the last one without an `else` part). Can you spot in what way do the two codes differ in their behaviour?

A more lively example

In the above example, we could have easily make it work without using exceptions:

In [21]:

```
from math import sqrt

a = float(input("a = "))
b = float(input("b = "))

if b == 0:
    print("Division by zero.")
else:
    result1 = a/b
    if result1 < 0:
        print("a/b does not have a real square root!")
    else:
        result2 = sqrt(result1)
        print("sqrt({} / {}) = {}".format(a, b, result2))
```

```
a = 17
b = 19
sqrt(17.0 / 19.0) = 0.9459053029269173
```

Obviously, the code is not very nice. A few more checks, and the indentation would get quite messy. Notice that there is an easy way around that by using a function (how?).

However, while the above code just got a bit "uglier" without exceptions, it is far more often that it would be much harder or even impossible to go without them. Consider the following, trivial yet very real example:

> Make sure that the user inputs proper real numbers. If they don't, do not crash the program, but warn the user and ask them to try again.

One way to deal with this problem is:

> Load a string with `s = input()` and then check if `s` contains a real number. If it does, convert it with `x = float(s)`; otherwise, report an error and go back to the input.

This seems simple enough. However, **the user's input is never to be trusted.** It can contain anything that is possible to type on the keyboard (and much, much more!), due to typos, errors, malicious intent, or some other reason.

If you try to write a code that will check if the input contains a valid "real" number, you'll see that it's not an easy thing to do (it gets easier with regular expressions (https://docs.python.org/3/library/re.html), but it's still not trivial). Here is what you'd have to consider:

- ignoring leading and trailing whitespaces (spaces, new lines, tabs,...),
- negative and positive numbers (with or without the prefixed +),
- with at most one decimal point (including without any),
- decimal (example: `-17.19`) and scientific (example: `-1.719e1`) format, with or without the exponent's sign.

So, while checking that the string `"-17.19"` can be converted to a (real) number seems easy, doing the same for `" \n \t -1.719e+1\f"`, while still disallowing any illegal inputs (like, for example, `" \n \t -1.719e+1\f1"` or `"1719e1.1"`), is much harder.

And all of this for just a simple input of a single real number!

Using the exceptions, this gets trivial:

In [22]:

```
while True:
    try:
        x = float(input("Input a real number: "))
    except ValueError:
        print("Shame on you! Don't you know what a real number is?!? :-P")
    else:
        break
print("Your real number:", x)
```

```
Input a real number: :-)
Shame on you! Don't you know what a real number is?!? :-P
Input a real number: 17
Your real number: 17.0
```

Note: When writing "real programs" (i.e., not just "school examples" or small utilities that only you will be using), you should always check that the input is valid. If it is not, handle it appropriately (print a message, request the input again, stop the program,... whatever you deem fit for that specific situation).

`finally` = "wrap it up"

One can also add a `finally` block to an exception handler:

```
try:
    do_something_1()
    do_something_2()
except SomeException:
    handle_exception()
else:
    no_error()
finally:
    wrap_it_all_up()
```

Here is what happens in some possible scenarios:

1. If no error occurs:

   ```
   do_something_1()
   do_something_2()
   no_error()
   wrap_it_all_up()
   ```

2. If `SomeException` occurs at the end of `do_something_1()`:

   ```
   do_something_1()
   handle_exception()
   wrap_it_all_up()
   ```

3. If `SomeOtherException` occurs at the end of `do_something_1()`:

   ```
   do_something_1()
   wrap_it_all_up()
   ```

   and `SomeOtherException` is still raised (as if `raise SomeOtherException(...)` was called after `wrap_it_all_up()`).

If it exists, **`finally` block is always executed before leaving the `try` block.**

`finally` with `break`, `continue`, and `return`

Yes, even when `break`, `continue`, or `return` is called within a `try` block, `finally` block will be executed before leaving.

In [23]:

```
n = int(input("x = "))
for i in range(n):
    try:
        if i*i == n:
            print(n, "is a complete square of", i)
            break
    finally:
        print("This is executed for i =", i)
```

```
x = 9
This is executed for i = 0
This is executed for i = 1
This is executed for i = 2
9 is a complete square of 3
This is executed for i = 3
```

However, `finally` itself catches no exceptions. For examle:

In [24]:

```
n = int(input("x = "))
for i in range(n):
    try:
        if i*i == n:
            print(n, "is a complete square of", i)
            raise Exception
    finally:
        print("This is executed for i =", i)
```

```
x = 9
This is executed for i = 0
This is executed for i = 1
This is executed for i = 2
9 is a complete square of 3
This is executed for i = 3

---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-24-700820e14c62> in <module>()
      4         if i*i == n:
      5             print(n, "is a complete square of", i)
----> 6             raise Exception
      7     finally:
      8         print("This is executed for i =", i)

Exception:
```

Some general rules

1. Catch as specific exception(s) as possible.
2. Put as little code as possible in the single exception block, to ensure that you catch exactly those exceptions that you want to catch.
3. Any clean-up should go in a `finally` block.

How deep does the rabbit hole go?

The beauty of the exceptions is that they allow us to control the flow over many levels of invocations, between several embedded loops, function calls, etc. Try the following example for $n = 2, 5, 11$:

In [25]:

```
def f(n, f, t):
    for i in range(1, n):
        for j in range(f, t):
            print("{} / {} = {}".format(i, j, i / j))

n = int(input("n = "))
if n % 2 == 0:
    print("The number is even, so let's call this function without handling any exceptio
ns:")
    f(n, -n, n)
else:
    print("The number is odd, so let's call this function:")
    try:
        if n > 7:
            f(n, -n, n)
        else:
            f(n, 1, n)
    except ZeroDivisionError:
        print("Whoops... You really do like dividing by zero!")
```

```
n = 11
The number is odd, so let's call this function:
1 / -11 = -0.09090909090909091
1 / -10 = -0.1
1 / -9 = -0.1111111111111111
1 / -8 = -0.125
1 / -7 = -0.14285714285714285
1 / -6 = -0.16666666666666666
1 / -5 = -0.2
1 / -4 = -0.25
1 / -3 = -0.3333333333333333
1 / -2 = -0.5
1 / -1 = -1.0
Whoops... You really do like dividing by zero!
```

Raising exceptions

Exceptions are in no way something magical that *just happens*. You can trigger them yourself, either to report an error or to achieve a fine control of your code's execution.

The correct term for triggering an exception is *raising* it, which is done with the keyword `raise` (other modern languages will use either `raise` or `throw`).

Consider the following problem:

> Write a function that takes an (integer) argument `n`, loads `n` real numbers, and returns the arithmetic mean of the loaded numbers.

This one should be trivial to write:

In [26]:

```
def arithmetic_mean(n):
    """
    Load `n` real numbers and return the arithmetic mean of the loaded numbers.
    """
    sum_ = 0
    for i in range(n):
        x = int(input("Number #{}: ".format(i+1)))
        sum_ += x
    return sum_ / n

print("Result:", arithmetic_mean(3))
```

```
Number #1: 17
Number #2: 19
Number #3: 23
Result: 19.666666666666668
```

This code will execute as expected. However, there is a designer flaw in it. Can you spot it?

What happens if we do this?

In [27]:

```
print("Result:", arithmetic_mean(0))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-27-e3171e379fdb> in <module>()
----> 1 print("Result:", arithmetic_mean(0))

<ipython-input-26-a1961cc5aec1> in arithmetic_mean(n)
      7         x = int(input("Number #{}: ".format(i+1)))
      8         sum_ += x
----> 9     return sum_ / n
     10
     11 print("Result:", arithmetic_mean(3))

ZeroDivisionError: division by zero
```

One could argue that the input is faulty and makes no sense, hence the error is expected. While this is true, and the code should do this, the error itself is confusing:

> ZeroDivisionError: division by zero

What division? The user of our function knows that they have to get an arithmetic mean, but they need not know how it is calculated. Imagine if the problem was more complex than a simple arithmetic mean: it would be perfectly OK for a user to not know the internal functioning of a function that you wrote.

Further, this will report no error:

In [28]:

```
print("Result:", arithmetic_mean(-17))
```

```
Result: -0.0
```

This result is a nonsense, in a way that an arithmetic mean is not defined for a negative number of numbers (actually, "-17 numbers" is a nonsense itself). One could argue that the result 0 makes some sense, but it is certainly different from the arithmetic mean of, for example, `{-17,-2,19}`. The user should be made aware of that.

So, how to deal with this?

Traditionally, the function would return an invalid value. For example, if we were computing the sum of digits, -1 would be an invalid result. However, there is no "invalid arithmetic mean" -- every real number is an arithmetic mean of an infinite number of sets of real numbers.

We could return `None`, but this solution is bad because it doesn't really imply an error. Unless the user checks specifically for `None`, they might never realize that there was an error. Any value can be compared with `None` (the result will be `False`, unless that value is also `None`). Also, returning `None` or its equivalent will not work in strongly typed languages (C++, for example).

The proper way to do this is to raise your own exception.

First we need to determine the type of an exception, so that the exception properly describes what has happened. We can always define our own exceptions, but usually we can pick among the existing ones (https://docs.python.org/3/library/exceptions.html).

Here, the problem is in the wrong value of `n`, which fits perfectly in the description of the `ValueError` (https://docs.python.org/3/library/exceptions.html#ValueError) type:

> *exception* `ValueError`
>
> Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError` (https://docs.python.org/3/library/exceptions.html#IndexError).

We are now ready to go:

In [29]:

```
def arithmetic_mean(n):
    """
    Load `n` real numbers and return the arithmetic mean of the loaded numbers.
    If `n` has an illegal value (i.e., `n <= 0`), the function rises a ValueError except
ion.
    """

    if n <= 0:
        raise ValueError("Arithmetic mean of zero or less numbers is undefined")

    sum_ = 0
    for i in range(n):
        x = int(input("Number #{}: ".format(i+1)))
        sum_ += x
    return sum_ / n

print("Result:", arithmetic_mean(3))
```

```
Number #1: 17
Number #2: 19
Number #3: 23
Result: 19.666666666666668
```

The previous illegal value example will now raise an exception:

In [30]:

```
print("Result:", arithmetic_mean(-17))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-30-3080ce387349> in <module>()
----> 1 print("Result:", arithmetic_mean(-17))

<ipython-input-29-1834822a092f> in arithmetic_mean(n)
      6
      7     if n <= 0:
----> 8         raise ValueError("Arithmetic mean of zero or less numbers i
s undefined")
      9
     10     sum_ = 0

ValueError: Arithmetic mean of zero or less numbers is undefined
```

We can now make a call like this, to report our own error message:

In [31]:

```
n = int(input("n = "))
try:
    print("Result:", arithmetic_mean(n))
except ValueError:
    print("The arithmetic mean is well defined only when n > 0")
```

```
n = -17
The arithmetic mean is well defined only when n > 0
```

or like this, to use the one defined in the function:

In [32]:

```
n = int(input("n = "))
try:
    print("Result:", arithmetic_mean(n))
except ValueError as e:
    print(e)
```

```
n = -17
Arithmetic mean of zero or less numbers is undefined
```

or like this, to treat "bad input" as if the result was zero:

In [33]:

```
n = int(input("n = "))
try:
    am = arithmetic_mean(n)
except ValueError:
    am = 0
print("Result:", am)
```

```
n = -17
Result: 0
```

Silently handling exceptions

How do we silently handle an error (i.e., prevent it from crashing a program, but not really do anything about it)?

This won't work (because `except` must be followed by at least one command):

In [34]:

```
try:
    x = 0 / 0
except ZeroDivisionError:

print("We're OK!")
```

```
  File "<ipython-input-34-6f51868040c3>", line 5
    print("We're OK!")
        ^
IndentationError: expected an indented block
```

This leads us to...

Doing nothing

A favourite activity of many people, doing nothing, is implemented in Python as the `pass` statement. It literally means "do nothing" (so, it's not a part of control flow in the strict sense) and it is used when the syntax requires at least one command, but there is none to be executed.

Here is an example:

In [35]:

```
n = int(input("n = "))
if n > 17:
    pass
else:
    print("Your number is smaller than or equal to 17.")
```

```
n = 19
```

Without `pass`, this would fail:

In [36]:

```
n = int(input("n ="))
if n > 17:

else:
    print("Your number is smaller than or equal to 17.")
```

```
  File "<ipython-input-36-bb95733acf99>", line 4
    else:
       ^
IndentationError: expected an indented block
```

Obviously, there is a better way to do the above:

In [37]:

```
n = int(input("n ="))
if n <= 17:
    print("Your number is smaller than or equal to 17.")
```

```
n =19
```

However, there are legitimate situations when we need to have empty blocks.

One such example can be "I need to add some code here, but I'll do it later":

```
if some_condition:
    pass # TODO: write the code to solve the problem if some_condition happens
else:
    a_general_solution()
```

Another example is defining our own exceptions:

```
class SomeNewError(Exception):
    pass
```

All we need here is a new type of exception (so it can be distinguished from other exceptions), but with no new functionality. This can now be used the same way we used `ValueError` above:

In [38]:

```
class SomeNewError(Exception):
    """
    Description of the error (never forget docstrings when defining something!)
    """
    pass

raise SomeNewError("This is a very informative message")
```

```
---------------------------------------------------------------------------
SomeNewError                              Traceback (most recent call last)
<ipython-input-38-19612a2b0833> in <module>()
      5     pass
      6
----> 7 raise SomeNewError("This is a very informative message")

SomeNewError: This is a very informative message
```

While the word "`Error`" in the name "`SomeNewError`" is not mandatory, it is a Python standard to use it for all exceptions that describe some error (see Exception Names in PEP 8 (https://www.python.org/dev/peps/pep-0008/#exception-names)).

The statement `pass` can also give us a solution to the question about handling exceptions silently:

In [39]:

```
try:
    x = 0 / 0
except ZeroDivisionError:
    pass

print("We're OK!")
```

```
We're OK!
```

Note that it is usually desirable to not just ignore an error, but it can come in handy.

Breaking out of several loops

As we have mentioned earlier, `break` will get you out of a single loop. However, using exceptions, you can break out of as many levels as you want. For example:

In [40]:

```
class BreakException(Exception):
    pass

n = int(input("n = "))
m = int(input("m = "))

for i in range(n):
    print("\ni =", i)
    try:
        for j in range(n):
            for k in range(n):
                s = i+j+k
                print("i + j + k = {} + {} + {} = {}".format(i, j, k, s))
                if s == m:
                    print("Stop!")
                    raise BreakException()
    except BreakException:
        pass
```

```
n = 3
m = 3

i = 0
i + j + k = 0 + 0 + 0 = 0
i + j + k = 0 + 0 + 1 = 1
i + j + k = 0 + 0 + 2 = 2
i + j + k = 0 + 1 + 0 = 1
i + j + k = 0 + 1 + 1 = 2
i + j + k = 0 + 1 + 2 = 3
Stop!

i = 1
i + j + k = 1 + 0 + 0 = 1
i + j + k = 1 + 0 + 1 = 2
i + j + k = 1 + 0 + 2 = 3
Stop!

i = 2
i + j + k = 2 + 0 + 0 = 2
i + j + k = 2 + 0 + 1 = 3
Stop!
```

Note: The name `BreakException` is just something we have chosen. It can be called any way we like, but it is highly desirable to make it descriptive.

Note that the code can almost always be *refactored* (i.e., rewritten) by converting its parts to functions and then breaking loops with `return`. For the above code, it would look like this:

In [41]:

```
n = int(input("n = "))
m = int(input("m = "))

def inner_loops(i):
    """
    Inner loops that sometimes need to be interrupted all at once.
    We are using `return` to achieve that effect.
    """
    for j in range(n):
        for k in range(n):
            s = i+j+k
            print("i + j + k = {} + {} + {} = {}".format(i, j, k, s))
            if s == m:
                print("Stop!")
                return

for i in range(n):
    print("\ni =", i)
    inner_loops(i)
```

```
n = 3
m = 3

i = 0
i + j + k = 0 + 0 + 0 = 0
i + j + k = 0 + 0 + 1 = 1
i + j + k = 0 + 0 + 2 = 2
i + j + k = 0 + 1 + 0 = 1
i + j + k = 0 + 1 + 1 = 2
i + j + k = 0 + 1 + 2 = 3
Stop!

i = 1
i + j + k = 1 + 0 + 0 = 1
i + j + k = 1 + 0 + 1 = 2
i + j + k = 1 + 0 + 2 = 3
Stop!

i = 2
i + j + k = 2 + 0 + 0 = 2
i + j + k = 2 + 0 + 1 = 3
Stop!
```