Errors

- Taking input from a user, and then validating it, is a rather big topic on its own that we do not want to touch.
- Nonetheless, your code should to some extent be prepared for thing to go wrong.
- E.g., what if `s=input()` is suppose to give a integer, the code does `int(s)`, but the input isn't integer? We get some funny error like:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '45.1'
```

- We will talk about Python's error messages later, but the users of your script do not wish to decipher that.

Error handling

- You could check if the string is in fact a number, as there's a function for that.
- It would look like this:

```
s=input("Give me an integer: ")
if s.isnumeric():
   i=int(s)
   print(i)
else:
   print("That is not an integer!")
```

- Good, but there may be other things that go wrong in the input that we did not catch.

- An alternative is the 'try first, deal with failure later' model: exceptions.
- This take the following form

```
s=input("Give me an integer: ")
try:
   i=int(s)
   print(i)
except:
   print("That is not an integer!")
```

- You can be more specific in the except on what kind of error you're catching, but let's not worry about that now.

Hands-on #4

Create a script that:

- Read two strings, call them `astr` and `bstr`
- Check if a number, if not error, else print the sum of `astr` and `bstr`.

Python's error messages

Because we cannot foresee every possible error, let's look at a typical uncaught Python error.

```
>>> print 17
  File "<stdin>", line 1
    print 17
          ^
SyntaxError: Missing parentheses in call to 'print'
```

Read the lines in the error messages carefully:

1. Something's up in line 1 in a file "`<stdin>`", i.e., the prompt.
2. The statement with the issue is printed, here, it is `print 17`.
3. The ^ more precisely pinpoints where there's an issue
4. The last line is most informative: there should have been parentheses in the call to 'print'.
   The type of this error is a 'SyntaxError'.

Python's error messages

Let's look at another error message:

```
>>> print(seventeen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'seventeen' is not defined
```

Read the lines in the error messages carefully:

1) *What's a traceback?*

   When the error occurs in the execution step, several function may be called before the error, and the traceback would show these.

2) Here, the error occurs in line 1 in a file "`<stdin>`", i.e., the prompt, but before a function has been called, i.e., on the "`<module>`" level.

3) Again, the last line is most informative: the variable seventeen has not been defined (yet?).
   The type of this error is a 'NameError'.

Python's error messages

Here's another one:

```
>>> a = 11
>>> b = '17'
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

What was going on here?

Section 4

Repetitions/Loops

Try again

- In the read-an-int example, it would be nice to start over if the user didn’t enter an integer.
- A ‘go to beginning’ statement does not exist in Python (no ’go-to’s in fact), but loops are.
- Loops are repetitions of a code block for different, given cases, or until a condition is fulfilled.
- So we could ‘loop’ (do the same thing over and over again) until the entered string is an integer.
- This would be a `while` loop.

  (The other kind of loop is a `for`, which we will see later)

While loop

- In the read-an-int example, it would be nice to start over if the user didn't enter an integer.
- We could 'loop' until the entered string is an integer.

This is one way:

```
haveint=False
while not haveint:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   haveint=True
  except:
   print("That is not an integer, try again!")
print(i)
```

- At the start of the while loop, `haveint` is checked, and Python enters the the code block that belongs to `while` (the "body of the loop")
- If `i=int(s)` succeeds, `haveint` is set to True.
- `haveint` is checked at the next iteration.
- Note that `print(i)` is outside the loop body.

Escaping the loop

- If the expression after `while` is not true after the loop body is executed, the loop stops.
- The loop can also be stopped at any time in the loop body with the `break` keyword.

In both cases, execution of the script continues with the next non-indented line of code.

So instead of:

```
haveint=False
while not haveint:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   haveint=True
  except:
   print("That is not an integer, try again!")
print(i)
```

We could also have used:

```
while True:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   break
  except:
   print("That is not an integer, try again!")
print(i)
```

Note: break stops the loop, but not the script. The `exit()` function can stop a script.

Hands-on #5

Create a script that:

- Reads two strings, call them `astr` and `bstr`
- Checks if they are numbers, if not, let user know which one is wrong, and let them enter both numbers again.
- If they both contain numbers, print the sum of integer values of `astr` and `bstr` and exit.