Python Fundamentals 1

Errors in Python PF1

Learning Objectives

1 What is an error?
2 To understand different types of errors
3 Identifying common errors
4 How to handle errors programmatically

Introduction

What are errors in Python? As with many programming languages, Python has built-in error reporting: a system whereby problems that stop a program from running as intended, are reported to the user, on the console. Errors occur when the Python interpreter encounters a line or syntax that it cannot understand, or cannot execute.

These errors fall into two broad categories:

- **syntax errors** - these happen when the rules of the Python language are broken (such as missing colons or brackets).

- **runtime errors** - these occur while a program is running, when otherwise valid code fails during its execution.

When an error occurs, Python generates an error message called a **traceback**. These tracebacks explain what went wrong to cause the error, and where in the code the problem occurred. Learning to read and understand these error messages is of paramount importance. When writing code, and testing its execution, these errors help to quickly pinpoint where the problems occur, allowing us to repair or **debug** the code with ease, and precision. Thus, they build into us as programmers, training sharper problem-solving strategies within us that lead us to writing more robust code.

Syntax errors

We have already defined these as errors that arise when Python's "grammar" or language rules have been broken. They prevent the program from even starting to run, because the interpreter cannot make sense of the code structure. Syntax errors are often the first type of error that new programmers encounter, and the good news is that they are usually easy to spot once you know what to look for.

Common examples of syntax errors include:

- **Missing or unmatched quotes** – forgetting to close a string with the same quotation mark you opened it with

- **Missing or unmatched parentheses** – leaving out a bracket, or using one too many

- **Missing colons** – forgetting the colon `:` that must follow an `if`, `elif`, `else` conditional (as we looked at, previously) or later on in the course, a `for` and `while` loop

- **Incorrect indentation** – not lining up code blocks with the correct number of spaces or tabs

- **Misspelled keywords** – for instance, typing `pritn` instead of `print`, or `esle` instead of `else`

- **Illegal characters** – including symbols or characters that Python does not recognise

When Python encounters one of these problems, it produces a `SyntaxError` message and highlights the part of the cod it cannot interpret. The caret `^` symbol in an error message often *points to the exact spot* where the interpreter got confused. It may not always be the true location of the mistake, but it is usually close by. Developing the habit of carefully reading error messages and checking for these common issues is the first step towards becoming fluent in **debugging** - a term used to describe spotting errors or 'bugs' in code that impede its execution.

We will generate each of the different types of `SyntaxError` in the cells, below:

```python
# Missing quotes around a string:

print("Gene symbol: BRCA1)
```

Run Code

Ready to run code!

```python
# Unmatched parentheses around a function's arguments:

print("Heart rate (bpm):", heart_rate
```

Run Code

Ready to run code!

```python
# Missing colon in a conditional statement:

temperature = 38.2

if temperature > 37.5
    print("Fever detected")
```

Run Code

Ready to run code!

```python
# Missing indentation in a conditional statement:

```

```python
wbc = 12000

if wbc > 11000:
print("Leukocytosis")  # This line should be indented (one tab, or four spaces)
```

Run Code

Ready to run code!

```python
# Misspelling:

bmi = 31.4

if bmi >= 30:
    print("Obese.")

esle:                          # In this line, 'else' is misspelled.
    print("Not obese.")
```

Run Code

Ready to run code!

```python
# Illegal characters:

glucose@level = 5.6   # The '@' character is not allowed in a variable name
```

Run Code

Ready to run code!

Upon reading the error messages, among the many things you will have noticed, is how informative they are. Not only do they diagnose the error and its location in your code, but they often also offer suggested fixes.

NOTE

For code with multiple errors, the Python interpreter will cease immediately upon hitting the **first error**. Thus, you wil only get error reporting on subsequent errors, once the first error has been fixed. This leads to the continuous struggle o a programmer where fixing one error just brings you more.

Runtime errors

Unlike syntax errors, which prevent completley prevent a program from running, **runtime errors** occur *while* the program is already running. Thus, a `RuntimeError` indicates that the code is written in a way that Python can understand *syntactically*, but when it tries to execute a particular line, something goes wrong along the way. Runtime errors are also referred to as **exceptions**, as they represent when the program's normal flow is interrupted, in some way They are among the most common errors you will encounter as you gain experience, and your programs build both in terms of complexity and interactivity.

A key feature of runtime errors is that they stop your program only *at the point where the problem occurs*. Everything *before* the error will have run successfully, but nothing after it will execute until the error is resolved.

The error message provides a *traceback* that explains what type of error occurred, together with the exact line number where the problem was triggered. A traceback is a unique type of error message in Python, where a report of all the steps the interpreter ran successfully up until the point of the error, are listed out. If and where applicable, this tracebac will tell you the file name, line number and the type of error which, again, informs the user as copiously as possible, sc that debugging is made easy.

Common types of runtime errors include:

- **`NameError`** – using a variable that has not been defined
- **`TypeError`** – performing an operation on the wrong data type (e.g. `"DNA" + 5`)
- **`ValueError`** – passing an invalid value to a function (e.g. `int("gene")`)
- **`ZeroDivisionError`** – attempting to divide a number by zero
- **`IndexError`** – trying to access a position outside the bounds of a list or string
- **`KeyError`** – looking up a dictionary key that does not exist
- **`ImportError`** / **`ModuleNotFoundError`** – when Python cannot locate the module you are trying to import

By learning to read and interpret these error messages, you can quickly trace back to the source of the problem and decide how best to correct it.

In the cells below, let's generate some runtime errors, and see a few of these examples, in action.

```python
# NameError – referring to a variable that has not been defined:

print(gene_sequence)
```

Run Code

Ready to run code!

```python
# TypeError – performing an operation on the wrong data type:

dna = "ATCG"

```

```python
print(dna + 5)    # Adding or concatenating a string + integer is not valid.
```

Run Code

Ready to run code!

```python
# ValueError – passing an invalid value to a function:

int("adenine")    # Cannot convert letters into an integer.
```

Run Code

Ready to run code!

```python
# ZeroDivisionError – attempting to divide a number by zero:

cell_count = 1000
replicates = 0

average = cell_count / replicates
```

Run Code

Ready to run code!

REMEMBER

The last three types of `RuntimeError` listed in this lesson, go beyond the lesson's scope. That is - they pertain to concepts and structures in Python that we have not covered, yet. For completeness, they are listed here, as they are important types of runtime errors, that you may encounter, further on in your Python learning.

Errors you'll see in the future

The errors you will see below are specific to some data structures in Python that we will be exploring in more detail in the next modules **PF2** and **PF3**. We'll give a brief description here for you to understand the error, however there will b more information in the future to understand the terminology properly.

Below is a **list** which is a type of data structure that can store multiple 'elements' or items (covered in **PF2**), which can be accessed individually using a technique or system called **indexing**.

In the case where an index is given that is greater than the actual number of elements in the list, you get a specific erro known as an `IndexError`, as demonstrated, below:

```python
# IndexError – trying to access a position outside the bounds of a list or string:

genes = ["BRCA1", "TP53", "EGFR"]

print(genes[5])    # The
```

Run Code

Ready to run code!

Similarly, this error is attributed to another type of associative data structure called a **dictionary** (covered in **PF3**), which contains multiple key:value pairs. A value is accessed by using its associated key, and if a key is used that is not in the dictionary, you get a specific `KeyError` as shown, below":

```python
# KeyError – looking up a dictionary key that does not exist

expression = {"BRCA1": 2.5, "TP53": 4.1}

print(expression["MYC"])    # "MYC" key not in dictionary
```

Run Code

Ready to run code!

Lastly, in Python you can import modules or packages of code to expand Python (covered in **PF3**). In short, these are authored collections of Python functions, methods and attributes that allow users to perform specific functions, not included in Python, by default. They, therefore, often have to be explicitly imported and installed to be implemented in your code. In the case that they haven't been, Python will throw a `ModuleNotFoundError`, as demonstrated, below:

```python
# ImportError / ModuleNotFoundError – module cannot be found
import genomics_toolkit    # not a real or installed module
```

Run Code

Ready to run code!

KEY TERMS

One potential area of confusion to be clear on, is the interchangeability of the terms **run time error** and **exception**. In Python, runtime errors are represented *as* **exceptions**: that is, as special events that interrupt the normal flow of a program when the interpreter encounters something it cannot handle during execution, such as dividing by zero or usin an undefined variable. As we have explored, each exception has a specific name (like `NameError` or `TypeError`) and is reported with a **traceback** that shows where the error occurred.

While people often use *runtime error* and *exception* interchangeably, it is more precise to say that **runtime errors in Python appear as exceptions**, with each exception type describing the exact problem.

Error handling

`try` and `except`:

So far, we have covered how exceptions are raised whenever Python encounters a problem during execution. By default, these exceptions stop the program *completely*, which is often helpful for debugging but not always desirable if you want your program to continue running, just to see how far it gets, for instance.

Python therefore provides an elegant way to *handle* exceptions using the keywords `try` and `except`. This allows you tc wrap a block of code in a `try` statement, and if an exception occurs, the program will *catch* it and execute the code inside the corresponding `except` block instead of crashing. This is not dissimilar to how a conditional statement works in that if an error occurs, an alternative 'route' through the program is provided, to try and circumvent it.

In this way, `try` and `except` give you tight control over how your program responds to errors, enabling you to display useful messages back to the console, take alternative actions or safely recover the point you reached without stopping the entire program. If your program is editing, writing or overwriting important data, for example, these recovery options can be invaluable in preventing irreparable damage from occurring.

Syntax

Similar to the conditional statements we looked at earlier in this lesson, the `try` keyword is followed by a colon `:`, and hitting return auto-indents the next line by a single tab space. All lines on this indentation level will then specify the code that is *wrapped* by the `try` keyword.

If we exit the scope of `try` on a new line, we can then use the `except` keyword, to specify alternative 'solution' code, should a particular runtime error or exception occur in the `try` block.

Run the code in the cell below. Python will raise a `ZeroDivisionError`:

```python
cell_count = 1000
replicates = 0

average = cell_count / replicates

print(f"Average cells per replicate: {average}")
```

Run Code

Ready to run code!

If we wrap the same code in a `try` statement, and use `except` to run code that prints an error message of our choosing t
the console, Python no longer raises an error. Instead, it runs the code wrapped in the `except` statement, as a 'solution'
should a specified `ZeroDivisionError` be raised.

As follows:

```python
# ZeroDivisionError

try:

    cell_count = 1000
    replicates = 0

    average = cell_count / replicates

    print(f"Average cells per replicate: {average}")

except ZeroDivisionError:

    print("Error: number of replicates cannot be zero.")
```

Run Code

Ready to run code!

Similarly, run the cell below, and enter your age as a word (string), and a `ValueError` will raise, because the `int` class expects numeric characters as input, only. In this case, instead of Python's internal error reporting displaying, the `except` statement's code will execute:

```python
# ValueError

try:

    age = int(input("Enter the patient's age: "))

    print(f"Recorded age: {age}")

except ValueError:

    print("Error: please enter the age as a number, not text.")
```

Run Code

Ready to run code!

Lastly, if we trigger Python to raise a `NameError` by calling an undefined variable, we can use an `except` statement to print our own customised error message:

```python
# NameError

```

```python
try:

    print(f"Expression level: {gene_expression}")

except NameError:

    print("Error: the variable 'gene_expression' has not been defined yet.")
```

Run Code

Ready to run code!

Summary

Understanding how to read, interpret and handle errors is one of the most important skills to develop early in your Python programming journey. As part of the 'algorithmic thinking' that our course focuses on developing, errors are crucial, in that they help us to develop the key problem-solving skills that lie at the heart of programming, and writing good, efficient and readable code.

In this lesson, we examined how Python handles **errors** and how we, as programmers, can interpret and prevent them. We distinguished between **syntax errors**, which occur when Python cannot parse our code due to incorrect structure, and **runtime errors**, which appear while the program is executing valid syntax, but encounters an unexpected situatior (such as dividing by zero or referencing an undefined variable). We also delved into how Python provides detailed **tracebacks** to help pinpoint the source and location of an error, and how understanding these messages is a key part of debugging. The lesson also introduced the use of **exceptions**, showing how Python raises specific error types (such as `ValueError`, `TypeError`, `IndexError`) depending on the problem encountered. Finally, we discuss strategies for **error handling** using `try` and `except` blocks, which allow code to fail in a controlled manner, offering workarounds for certain scenarios. Together, these ideas form a foundation for writing robust, maintainable and legible Python code.

Previous