16

A Chi-Squared Case Study

In this bonus chapter, we'll work through a case study that performs some statistical analysis of a dataset to determine if the data reveals anything potentially significant.

We'll look at a common statistical decision. The decision is described in detail at `http://www.itl.nist.gov/div898/handbook/prc/section4/prc45.htm`.

Background

We'll use the χ^2, **chi-squared**, test to see whether or not data is distributed randomly. To make this decision, we'll need to compute an expected distribution and compare the observed data to our expectations. A significant difference means there's something that needs further investigation. An insignificant difference means we can use the null hypothesis that there's nothing more to study: the differences are simply random variation.

We'll start with some backstory—some details that are not part of the case study, but often feature in an **exploratory data analysis** (**EDA**) application. We need to gather the raw data and produce a useful summary that we can analyze.

The data comes from quality assurance checks while producing silicon wafers. The defect data is collected in a database. We may use SQL queries to extract defect details for further analysis. For example, a query could look like this:

```
SELECT SHIFT, DEFECT_CODE, SERIAL_NUMBER FROM some_tables;
```

The output from this query could be a `.csv` file with individual defect details:

```
shift,defect_code,serial_number
1,None,12345
1,None,12346
1,A,12347
1,B,12348
```

The actual data includes thousands of wafers.

There are three operating shifts and four defect codes spread around the collection of samples. Most wafers are defect-free. The goal is to find any pattern to the defects; this can help engineers locate the root cause.

We need to summarize the preceding data. We may summarize at the SQL query level using the `COUNT` and `GROUP BY` statements. We may also summarize at the Python-application level. While a pure database summary is often described as being more efficient, this isn't always true. In some cases, using a Python application to summarize a simple extract of raw data can be faster than a summary created by a database server. If performance is important, both alternatives must be measured, rather than hoping that the database is the fastest.

The goal is to create summary data with three attributes:

- The shift
- The type of defect
- A count of defects observed

The summary data needs to look like this:

```
shift,defect_code,count
1,A,15
2,A,26
3,A,33
```

The actual data will show all of the 12 combinations of three shifts and four defect types.

In the next section, we'll focus on reading the raw data to create summaries. This is the kind of context in which Python is particularly powerful: working with raw source data.

Once we have summaries, we need to compare the observed shift and defect counts with an expected value. If the difference between observed counts and expected counts can be attributed to random fluctuation, we have to accept the null hypothesis that nothing interesting is going wrong. If, on the other hand, the numbers don't fit with random variation, then we have a problem that requires further investigation.

Filtering and reducing the raw data with a Counter object

We'll represent the essential defect counts as a `collections.Counter` parameter. We will build counts of defects by shift and defect type from the detailed raw data. Here's the code that reads some raw data from a `.csv` file:

```
from typing import TextIO, NamedTuple, TypeAlias
import csv
from collections import Counter

class Defect(NamedTuple):
    shift: str  # domain is '1', '2', '3'
    defect_type: str  # domain is 'A', 'B', 'C', 'D'
    serial_number: str

Shift_Type: TypeAlias = tuple[str, str]
```

```
def defect_reduce(input_file: TextIO) -> Counter[Shift_Type]:
    rdr = csv.DictReader(input_file)
    defect_iter = (Defect(**row) for row in rdr)
    shift_type_iter = (
        (row.shift, row.defect_type)
        for row in defect_iter if row.defect_type)
    tally = Counter(shift_type_iter)
    return tally
```

The `defect_reduce()` function will create a dictionary reader based on an open file provided through the input parameter. Each row of input creates an instance of the `Defect` named tuple. Because the column names in the file are valid Python variable names, it's easy to transform the dictionary into a named tuple object. This allows us to use slightly simpler syntax to refer to items within the row. Specifically, the next generator expression uses references such as `row.shift` and `row.defect_type` instead of the slightly more wordy `row['shift']` or `row['defect_type']` references.

We can use a more complex generator expression to do a map-filter combination. We'll filter each row to ignore rows with no defect code. For rows with a defect code, we're mapping an expression that creates a two-tuple from the `row.shift` and `row.defect_type` references.

In some applications, the filter won't be a trivial expression such as `row.defect_type`. It may be necessary to write a more sophisticated condition. In this case, it may be helpful to use the `filter()` function to apply the complex condition to the generator expression that provides the data.

Given a generator that will produce a sequence of `(shift, defect_type)` tuples, we can summarize them by creating a `Counter` object from the generator expression. Creating this `Counter` object will process the lazy generator expressions, which will read the source file, extract fields from the rows, filter the rows, and summarize the counts.

We've provided a `qa_data.csv` file in the repository of code for this book.

We'll use the `defect_reduce()` function to gather and summarize the data as follows:

```
>>> from pathlib import Path

>>> source_path = Path.cwd() / "qa_data.csv"
>>> with source_path.open() as input:
...     defect_counts = defect_reduce(input)
```

We can open the file, gather the defects, and display them to be sure that we've properly summarized by shift and defect type. Since the result is a `Counter` object, we can combine it with other `Counter` objects if we have other sources of data.

The defect summary counts look like this:

```
>>> from pprint import pprint

>>> pprint(defect_counts)
Counter({('3', 'C'): 49,
         ('1', 'C'): 45,
         ('2', 'C'): 34,
         ('3', 'A'): 33,
         ('2', 'B'): 31,
         ('2', 'A'): 26,
         ('1', 'B'): 21,
         ('3', 'D'): 20,
         ('3', 'B'): 17,
         ('1', 'A'): 15,
         ('1', 'D'): 13,
         ('2', 'D'): 5})
```

We have defect counts organized by shift (`'1'`, `'2'`, or `'3'`) and defect types (`'A'` through `'D'`). Using alternative inputs of summarized data is left as an exercise for the reader. This

reflects a common use case where data is available at the summary level.

Once we've read the data, the next step is to develop two probabilities so that we can properly compute expected defects for each shift and each type of defect. We don't want to divide the total defect count by 12, since that doesn't reflect the actual deviations by shift or defect type. The shifts may be more or less equally productive. The defect frequencies are certainly not going to be similar; we expect some defects to be very rare and others to be more common.

Computing sums with a Counter object

We need to compute the probabilities of defects by shift and defects by type. To compute the expected probabilities, we need to start with some simple sums. The first is the overall sum of all defects, which can be calculated by executing the following command:

```
>>> total = sum(defect_counts.values())
>>> total
309
```

This is done directly from the values in the `Counter` or dictionary object assigned to the `defect_counts` variable. This will show that there are 309 total defects in the sample set.

We need to get defects by shift as well as defects by type. This means that we'll extract two kinds of subsets from the raw defect data. The *by-shift* extract will use just one part of the `(shift, defect_type)` key in the `Counter` object. The *by-type* extract will use the other half of the key pair.

This suggests a higher-order function to summarize a `Counter` object with a multi-part key by a subset of the key. We're summarizing the original multi-dimensional grouping to use fewer dimensions. This function can apply a filter to each key to extract the relevant subset. Here's a function definition we can use for this:

```
from collections.abc import Callable
from typing import TypeAlias

ShiftDefect: TypeAlias = tuple[str, str]
SourceCounter: TypeAlias = Counter[ShiftDefect]

def summarize_by(
        subset_key: Callable[[ShiftDefect], str],
        source: SourceCounter
) -> Counter[str]:
    grouped_iter = (
        Counter({subset_key(k): v})
        for k, v in source.items()
    )
    return sum(
        grouped_iter,
        Counter()
    )
```

We summarize the initial `Counter` object by creating and summing additional `Counter` objects. Each is extracted from the initial set of the `Counter` objects assigned to the `defect_counts` variable.

We can't use the default initial value of 0 for the `sum()` function. We must provide an empty `Counter()` object as an initial value.

Here are the by-shift and by-defect-type summaries:

```
>>> shift_totals = summarize_by(lambda s_d: s_d[0], defect_counts)
>>> type_totals = summarize_by(lambda s_d: s_d[1], defect_counts)
```

The shift summary totals look like this:

```
>>> shift_totals
Counter({'3': 119, '2': 96, '1': 94})
```

The defect type summaries look like this:

```
>>> type_totals
Counter({'C': 128, 'A': 74, 'B': 69, 'D': 38})
```

We've kept the summaries as `Counter` objects, rather than creating simple `dict` objects or possibly even list instances. We'll generally use them as simple dicts from this point forward.

Computing probabilities from Counter objects

We've read the data and computed summaries in two separate steps. In some cases, we may want to create the summaries while reading the initial data. This is an optimization that may save a little bit of processing time. We could write a more complex input reduction that emitted the grand total, the shift totals, and the defect type totals. These `Counter` objects would be built one item at a time.

We've focused on using the `Counter` instances, because they seem to allow us flexibility. Any changes to the data acquisition will still create `Counter` instances and won't change the subsequent analysis.

The probability of a defect—by shift or by defect type—is the observed count of defects out of the total population of defects. This is a fraction, and it's often helpful to work with fractional values as Python `Fraction` objects to keep as much precision as possible. Converting to float values can introduce small errors because of the way float values can involve truncation.

As with the summaries, we have a common function with two small variants. One will summarize the defects by shift, the other by defect type. This will regroup the overall defect counts from a two-dimensional value—shift and defect type—to two separate one-

dimensional values.

This is often presented in a tabular form with summaries at the end of the rows or below each column. We'll define a reusable function that can produce either row or column totals. Here's how we can compute the probabilities of defect by shift and by defect type:

```
from fractions import Fraction

def probability_by(
        subset_key: Callable[[ShiftDefect], str],
        source: SourceCounter
) -> dict[str, Fraction]:
    total = sum(source.values())
    sub_totals = summarize_by(subset_key, source)
    return {
        k: Fraction(sub_totals[k], total)
        for k in sub_totals
    }
```

This `probability_by()` function uses the `summarize_by()` function to compute a subtotal by shift or by defect type. It then computes fractional values for each dimension's individual contribution to the overall total. Because these are fractions, the total will always be calculated correctly without the possible truncation problems associated with float values.

This could be restated as a composition of two functions. Instead of directly evaluating the `summarize_by()` function, this could work with the results of the `summarize_by()` function. We've left the alternative design as an exercise for the reader.

The following example shows how we use these functions to produce the summary fractions:

```
>>> P_shift = probability_by(lambda s_d: s_d[0], defect_counts)
>>> P_type = probability_by(lambda s_d: s_d[1], defect_counts)
```

We've created two mappings: `P_shift` and `P_type`. The `P_shift` dictionary maps a shift to a `Fraction` object, showing the shift's contribution to the overall number of defects.

Similarly, the `P_type` dictionary maps a defect type to a `Fraction` object, showing the defect type's contribution to the overall number of defects.

We've elected to use `Fraction` objects to preserve all of the precision of the input values. When working with counts like this, we may get probability values that make more intuitive sense to people reviewing the data.

The `P_shift` and `P_type` data looks like this:

```
>>> from pprint import pprint

>>> pprint(P_shift, width=64)
{'1': Fraction(94, 309),
 '2': Fraction(32, 103),
 '3': Fraction(119, 309)}

>>> pprint(P_type, width=64)
{'A': Fraction(74, 309),
 'B': Fraction(23, 103),
 'C': Fraction(128, 309),
 'D': Fraction(38, 309)}
```

A value such as `32/103` or `96/309` does not suffer from float truncation. Fractions may also be more meaningful to some people than decimal values like `0.3106`. We can easily get float values from `Fraction` objects, as we'll see later.

The shifts all seem to be approximately at the same level of defect production. The defect types vary, which seems typical. It appears that the defect `C` is a relatively common problem, whereas the defect `D` is much less common. Perhaps the second defect requires a more complex situation to arise in the manufacturing process.

Computing expected values and displaying a contingency table

The expected defect production is a combined probability. We'll compute the shift defect probability multiplied by the probability based on defect type. This will allow us to compute all 12 probabilities from all combinations of shift and defect type. We can weight these with the observed numbers and compute the detailed expectation for defects.

The following code calculates expected values:

```
def expected(source: SourceCounter) -> dict[ShiftDefect, Fraction]:
    total = sum(source.values())
    P_shift = probability_by(lambda s_d: s_d[0], source)
    P_type = probability_by(lambda s_d: s_d[1], source)
    return {
        (s, t): P_shift[s] * P_type[t] * total
        for t in sorted(P_type)
            for s in sorted(P_shift)
    }
```

The `expected()` function creates a dictionary that parallels the initial `defect_counts` object. This dictionary will have a sequence of two-tuples with keys and values. The keys will be two-tuples of shift and defect type. This dictionary is built from a generator expression that explicitly enumerates all combinations of keys from the `P_shift` and `P_type` dictionaries.

The value of the expected dictionary looks like this:

```
>>> from pprint import pprint

>>> pprint(expected(defect_counts))
{('1', 'A'): Fraction(6956, 309),
 ('1', 'B'): Fraction(2162, 103),
 ('1', 'C'): Fraction(12032, 309),
 ('1', 'D'): Fraction(3572, 309),
 ('2', 'A'): Fraction(2368, 103),
```

```
 ('2', 'B'): Fraction(2208, 103),
 ('2', 'C'): Fraction(4096, 103),
 ('2', 'D'): Fraction(1216, 103),
 ('3', 'A'): Fraction(8806, 309),
 ('3', 'B'): Fraction(2737, 103),
 ('3', 'C'): Fraction(15232, 309),
 ('3', 'D'): Fraction(4522, 309)}
```

Each item of the mapping has a key with both a shift number and a defect type code. Each key is associated with a `Fraction` value showing the probability of a defect based on shift time and defect type multiplied by the overall number of defects. Some of the fractions are simplified; for example, a value of `6624/309` was simplified to `2208/103`.

Large numbers are awkward when displayed as proper fractions. Displaying large numbers as float values is often easier. Small values (such as probabilities) are sometimes easier to understand as fractions.

We'll print the observed and expected times in pairs. This will help us visualize the data. We'll create a contingency table to help summarize what we've observed and what we expect.

We can use the `rich` package to incorporate rules to make the table easier to read. The following function displays the relevant table:

```
from rich.table import Table
from rich.console import Console

def contingency_table(expected: dict[ShiftDefect, Fraction],
defect_counts: Counter[ShiftDefect]) -> None:
    total = sum(defect_counts.values())
    shift_totals = summarize_by(lambda s_d: s_d[0], defect_counts)
    type_totals = summarize_by(lambda s_d: s_d[1], defect_counts)

    table = Table(title="Contingency Table")
    table.add_column("shift")
```

```
    for defect_type in sorted(type_totals):
        table.add_column(f"{defect_type} obs")
        table.add_column(f"{defect_type} exp")
    table.add_column("total")

    for s in sorted(shift_totals):
        row = [f"{s}"]
        for t in sorted(type_totals):
            row.append(f"{defect_counts[s,t]:3d}")
            row.append(f"{float(expected[s,t]):5.2f}")
        row.append(f"{shift_totals[s]:3d}")
        table.add_row(*row)

    footers = ["total"]
    for t in sorted(type_totals):
        footers.append(f"{type_totals[t]:3d}")
        footers.append("")
    footers.append(f"{total:3d}")
    table.add_row(*footers)

    console = Console()
    console.print(table)
```

The output can be a bit too wide to typeset well. The following example should suggest the content of the table:

```
>>> contingency_table(expected(defect_counts), defect_counts)
                            Contingency Table
```

shift	A obs	A exp	B obs	B exp	C obs	C exp	D obs	D exp	tot.
1	15	22.51	21	20.99	45	38.94	13	11.56	94
2	26	22.99	31	21.44	34	39.77	5	11.81	96
3	33	28.50	17	26.57	49	49.29	20	14.63	119
total	74		69		128		38		309

This shows 12 pairs of cells. Each pair has the observed number of defects and an expected number of defects. Each row is for a shift, and ends with the shift totals. Each pair of

columns is for a defect type, and has a footer with the defect totals.

In some cases, we might export this summarized data in CSV notation and build a spreadsheet. In other cases, we may build a Markdown or HTML version of the contingency table for publication. For more information on Markdown, see `https://www.markdownguide.org/`.

A contingency table such as this one can clarify the comparison between observed and expected values. From this, a chi-squared test can help decide if the observed values match the expected values. This will help us decide whether the data is random or whether there's something that deserves further investigation.

Computing the chi-squared value

The χ^2 value is based on $\sum_i \frac{(e_i - o_i)^2}{e_i}$, where the e_i values are the expected values and the o_i values are the observed values. In our case, we have two dimensions, shift, s, and defect type, t, which leads to the following computation:

$$\chi^2 = \sum_s \sum_t \frac{(e_{st} - o_{st})^2}{e_{st}}$$

We can compute the specified formula's value as follows:

```
def chi2(defect_counts: Counter[ShiftDefect]) -> Fraction:
    total = sum(defect_counts.values())
    shift_totals = summarize_by(lambda s_d: s_d[0], defect_counts)
    type_totals = summarize_by(lambda s_d: s_d[1], defect_counts)

    expected_counts = expected(defect_counts)

    diff_sq_e: Callable[[Fraction, int], Fraction] = (
        lambda e, o: (e - o) ** 2 / e
    )

    chi2 = sum(
        diff_sq_e(expected_counts[s, t], defect_counts[s, t])
```

```
        for s in shift_totals
        for t in type_totals
    )
    return Fraction(chi2)  # convert any int result to Fraction
```

We've defined a small lambda, `diff_sq_e`, to help us optimize the calculation. This allows us to execute the `expected_counts[s,t]` and `defect_counts[s,t]` attributes just once, even though the expected value is used in two places.

Here's how this `chi2` function works to compute a χ^2 value for our defect count data. We can compute a float value or a fraction value:

```
>>> round(float(chi2(defect_counts)), 2)
19.18
>>> chi2(defect_counts).limit_denominator(20)
Fraction(326, 17)
```

For this dataset, the final χ^2 value is 19.18. There are a total of six degrees of freedom based on three shifts and four defect types. A chi-squared table shows us that any value below 12.5916 would suggest 1 chance in 20 of the data being truly random. Since our value is 19.18, the data is unlikely to be random. Something is wrong that warrants deeper investigation.

Online chi-squared tables include this one from the US National Institute of Standards and Technology (NIST): `https://www.itl.nist.gov/div898/handbook/eda/section3/eda3674.htm`.

The cumulative distribution function shows that a value of 19.18 has a probability of the order of 0.00387: about 4 chances in 1,000 of being random. The next step in the overall analysis is to design a follow-up study to discover the details of the various defect types and shifts. We'll need to see which independent variable has the biggest correlation with defects and continue the analysis. This work is justified because the χ^2 value indicates that

the effect is not simple random variation.

Summary

In the case study presented in this chapter, we looked at the advantage of using Python for exploratory data analysis—the initial data acquisition including a little bit of parsing and filtering. In some cases, a significant amount of effort is required to normalize data from various sources. This is a task at which Python excels.

The calculation of an χ^2 value involved three intermediate `sum()` functions: two from generator expressions, and a final generator expression to create a dictionary with expected values. After these three, a final `sum()` function created the statistic. In under a dozen expressions, we created a sophisticated analysis of data that will help us accept or reject the null hypothesis.

Using a functional approach, we can write succinct and expressive programs that accomplish a great deal of processing. Python isn't a properly functional programming language. For example, we're required to use some imperative programming techniques. This limitation forces us away from purely functional recursions. We gain some performance advantage, since we're forced to optimize tail recursions into explicit loops.

We saw numerous advantages of adopting Python's hybrid style of functional programming. In particular, Python's higher-order functions and generator expressions give us a number of ways to write high-performance programs that are often quite clear and simple.

Exercises

This chapter's exercises are based on code available from Packt Publishing on GitHub. See `https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Editi` on.

In some cases, the reader will notice that the code provided on GitHub includes partial solutions to some of the exercises. These serve as hints, allowing the reader to explore alternative solutions.

In many cases, exercises will need unit test cases to confirm they actually solve the problem. These are often identical to the unit test cases already provided in the GitHub repository. The reader should replace the book's example function name with their own solution to confirm that it works.

Reading summary data

In the *Filtering and reducing the raw data with a Counter object* section, we looked at how to read raw data and create a useful `Counter` object with summary counts organized by shift and defect type.

As an alternative to reading all of the raw data, we can look at processing only the summary counts. We want to create a `Counter` object similar to that example; this will have defect counts as a value, with a key of shift and defect code.

The data will be a CSV file (or a database query result) with three columns: shift, defect type, and a count. These need to be grouped by a two-tuple of shift and defect type.

Write a `defect_summary()` function that can work with a source of data and produce the expected `Counter` object.

The results of this function will match the detail summary shown previously. The online repository for this book only has the raw detailed data, so the first step to testing this function is creating and writing a summary file.

Once this summary data is available as a CSV-formatted file, the function to read the summaries can be tested.

Revise the probability_by() and summarize_by() functions

In the *Computing probabilities from Counter objects* section of this chapter, we defined a function, `probability_by()`, that evaluated the `summarize_by()` function directly. There's some overlap between the filtering rules used by these functions. Both functions use a `subset_key: Callable[[ShiftDefect], str]` function to identify relevant data.

The `probability_by()` function could be restated as a composition of two functions.

Instead of directly evaluating the `summarize_by()` function for each row, define a `probability_by()` function that works with the subtotals created by the `summarize_by()` function. This means the subtotals must be a parameter to the revised `probability_by()` function. It also means the functions using this revised function must be changed to reflect the new set of parameters.

Fraction vs. float

Throughout this case study, an emphasis was placed on using `Fraction` objects instead of `float`. This leads to extremely precise answers, perhaps more precise than is truly needed.

Rewrite the functions here to use `float` instead of `Fraction` objects. How much faster is the computation? Is this a helpful improvement?