Module 2: Collections, Loops, and Functions

PUBHLT 0411 · Python for Public Health Data Analysis

Soumik Purkayastha
Dept. of Biostatistics and Health Data Science, University of Pittsburgh

October 15, 2026

Forty-nine months.

We will examine 49 months of COVID-19 death data across all 67 counties in Pennsylvania.

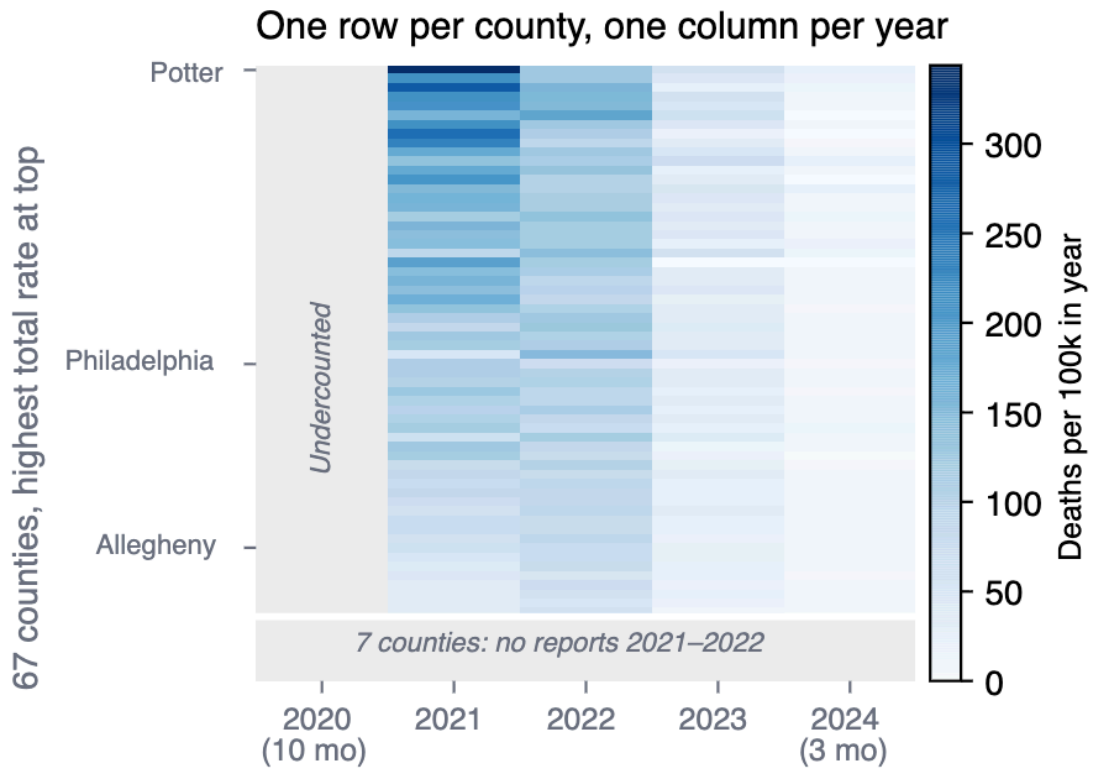
Pennsylvania recorded COVID-19 deaths across 49 months and 67 counties

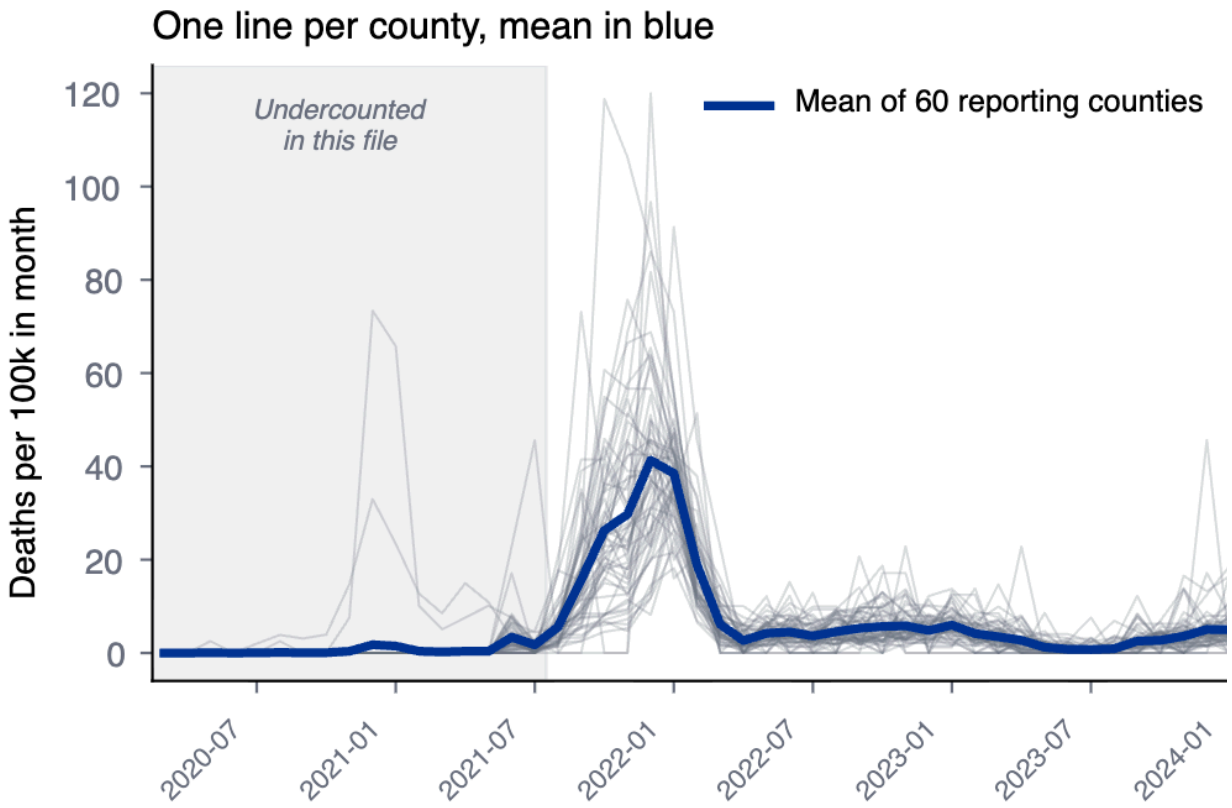
That is $49 \times 67 = 3,283$ county-months of data.

> **The question this session opens**
>
> Identifying the deadliest month requires examining every month and retaining the largest. Performed by hand, this is 49 comparisons for the state and 3,283 for the counties.
>
> **This session establishes the machinery that performs the comparison.** A collection to hold the values, a loop to walk them, and a function to package the work for reuse.

Required from Module 1: `if` / `elif` / `else`, indentation, lists, `.append()`, `+=`.

Two views of the same 3,283 county-months





Source: PA DOH, monthly COVID-19 deaths by county

Each cell of the heatmap and each point on a grey line is one county-month. Producing either figure requires walking all 3,283 of them.

A dictionary maps keys to values

A list is indexed by position. A dictionary is indexed by a **key** chosen by the author.

Python Code | Start Over | Run Code

```
1  covid_deaths = {
2      "2021-08": 555,
3      "2021-09": 1248,
4      "2021-10": 2035,
5      "2021-11": 2117,
6      "2021-12": 3365,
7      "2022-01": 4294,
8      "2022-02": 1819,
9  }
10
11 print(covid_deaths["2021-12"])
12 print(len(covid_deaths))
```

Each entry is a `key: value` pair. **Keys are unique**; assigning to an existing key replaces its value.

> **R analogue**
>
> A named list, or a two-column lookup table.

A missing key raises `KeyError`

Python Code | Start Over | Run Code

```
1  covid_deaths = {"2021-12": 3365, "2022-01": 4294}
2
3  try:
4      print(covid_deaths["2022-06"])
5  except KeyError as e:
6      print(f"{type(e).__name__}: {e}")
```

⚠ **`KeyError` — the key is not present**

`KeyError: '2022-06'`

Subscripting a dictionary with an absent key raises immediately rather than returning an empty value. The exception names the offending key.

Resolution: `.get(key, default)` returns the default instead of raising — `covid_deaths.get("2022-06", 0)` evaluates to `0`.

Python Code | Start Over | Run Code

```
1  covid_deaths = {"2021-12": 3365, "2022-01": 4294}
2
3  print(covid_deaths.get("2022-06", 0))
4  print(covid_deaths.get("2021-12", 0))
```

`for` walks every element of a collection

Python Code | Start Over | Run Code

```
1  covid_deaths = {
2      "2021-11": 2117,
3      "2021-12": 3365,
4      "2022-01": 4294,
5  }
6
7  for month in covid_deaths:
8      print(month, covid_deaths[month])
```

Iterating a dictionary yields its **keys**. The colon and the indented block follow the same rule as `if`.

Python Code | Start Over | Run Code

```
1  covid_deaths = {
2      "2021-11": 2117,
3      "2021-12": 3365,
4      "2022-01": 4294,
5  }
6
7  for month, deaths in covid_deaths.items():
8      print(f"{month}: {deaths} deaths")
```

`.items()` yields both parts at once, which is the idiomatic form.

Three views of a dictionary

`.keys()` yields the labels, `.values()` yields the quantities, `.items()` yields both.

`range()` generates a sequence of integers

Python Code | Start Over | Run Code

```
1 for i in range(5):
2     print(i)
```

Python Code | Start Over | Run Code

```
1 for year in range(2020, 2024):
2     print(year)
```

1. `range(n)` yields `0` through `n - 1` — **n values, starting at zero**
2. `range(a, b)` yields `a` through `b - 1` — the stop is excluded, as in slicing

> ⚠ **`range()` excludes its endpoint**
>
> `range(2020, 2024)` yields 2020, 2021, 2022, and 2023. Reaching 2024 requires `range(2020, 2025)`.

The accumulator pattern builds a result across iterations

Initialize a variable **before** the loop; update it **inside**.

Python Code | Start Over | Run Code

```python
covid_deaths = {
    "2021-08": 555, "2021-09": 1248, "2021-10": 2035, "2021-11": 2117,
    "2021-12": 3365, "2022-01": 4294, "2022-02": 1819,
}

total_deaths = 0

for month, deaths in covid_deaths.items():
    total_deaths += deaths

print(f"Total across {len(covid_deaths)} months: {total_deaths}")
print(f"Monthly average: {total_deaths / len(covid_deaths):.1f}")
```

Initialization belongs outside the loop

A counter initialized inside the loop resets on every iteration and finishes holding only the final value.

The maximum requires remembering the best so far

Python Code | Start Over | Run Code

```python
covid_deaths = {
    "2021-08": 555, "2021-09": 1248, "2021-10": 2035, "2021-11": 2117,
    "2021-12": 3365, "2022-01": 4294, "2022-02": 1819,
}

worst_month = ""
worst_deaths = 0

for month, deaths in covid_deaths.items():
    if deaths > worst_deaths:
        worst_deaths = deaths
        worst_month = month

print(f"Worst month: {worst_month} with {worst_deaths} deaths")
```

The loop retains a candidate and replaces it whenever a larger value appears. **This is the same procedure a person performs by hand, stated precisely.**

Checkpoint: the comparison is inside the wrong block

Objective: count the months exceeding 2,000 deaths.

Python Code | Start Over | Run Code

```python
covid_deaths = {
    "2021-08": 555, "2021-09": 1248, "2021-10": 2035, "2021-11": 2117,
    "2021-12": 3365, "2022-01": 4294, "2022-02": 1819,
}

high_months = 0

for month, deaths in covid_deaths.items():
    if deaths > 2000:
        high_months += 1
        print(f"{month}: {deaths}")
        print(f"Running count: {high_months}")

print(f"Months above 2000: {high_months}")
```

⚠ **Indentation determines what runs when**

The final `print()` is outside the loop and reports the total once, correctly. The running count is inside the `if` and prints on every match.

Moving the final line inside the loop would report a partial count as though it were the answer — with no error raised.

`while` repeats until a condition fails

A `for` loop runs a known number of times. A `while` loop runs until something changes.

Python Code | Start Over | Run Code

```python
covid_deaths = {
    "2021-08": 555, "2021-09": 1248, "2021-10": 2035, "2021-11": 2117,
    "2021-12": 3365, "2022-01": 4294, "2022-02": 1819,
}

months = list(covid_deaths.keys())
cumulative = 0
position = 0

while cumulative <= 10000:
    month = months[position]
    cumulative += covid_deaths[month]
    position += 1

print(f"Cumulative deaths passed 10,000 during {month}: {cumulative}")
```

⚠ A condition that never becomes false runs forever

Every `while` loop requires something inside it that moves the condition toward failure. Omitting `position += 1` produces a loop that never terminates.

`while True:` with `break` is an exit-controlled loop

Python Code | Start Over | Run Code

```
covid_deaths = {
    "2021-08": 555, "2021-09": 1248, "2021-10": 2035,
    "2021-11": 2117, "2021-12": 3365, "2022-01": 4294,
}

months = list(covid_deaths.keys())
position = 0

while True:
    month = months[position]
    if covid_deaths[month] > 3000:
        print(f"First month above 3000: {month}")
        break
    position += 1
```

`break` exits the loop immediately. `continue` skips to the next iteration without leaving.

The loop that tests at the bottom

`while True:` with `break` runs the body first and decides afterward. This is Python's equivalent of a do-while loop, and it suits searches that stop at the first match.

Functions.

The same calculation, written once and reused — instead of copied and edited every time a new county arrives.

`def` packages a computation under a name

Python Code | Start Over | Run Code

```
1  def rate_per_100k(deaths, population):
2      """Return deaths per 100,000 people."""
3      return deaths / population * 100000
4
5
6  print(rate_per_100k(401, 143786))
7  print(rate_per_100k(208, 157795))
8  print(f"{rate_per_100k(4052, 1550542):.1f}")
```

1. `def` names the function; the parameters are named in parentheses
2. The **docstring** — a string on the first line — states what the function returns
3. `return` sends a value back to the caller

> **A function without `return` returns `None`**
>
> Printing a value inside a function displays it; it does not hand it back. Only `return` produces a value the caller can store or reuse.

A function may return more than one value

Python Code | Start Over | Run Code

```python
def summarize_county(county, deaths, population):
    """Return the county's rate and a classification against the state rate."""
    rate = deaths / population * 100000
    if rate > 212.3:
        classification = "Above the state rate"
    else:
        classification = "At or below the state rate"
    return rate, classification


rate, label = summarize_county("Schuylkill", 401, 143786)

print(f"Schuylkill: {rate:.1f} per 100,000 — {label}")
```

Returning several values produces a **tuple**, and assigning to several names unpacks it.

A tuple is an immutable sequence

Python Code | Start Over | Run Code

```
1 county_record = ("Schuylkill", 401, 143786)
2
3 county, deaths, population = county_record
4
5 print(county, deaths, population)
6 print(county_record[0])
7 print(len(county_record))
```

Tuples index and slice exactly as lists do. **They cannot be modified after creation** — there is no `.append()`.

Tuples signal fixed structure

A list holds a variable number of similar things. A tuple holds a fixed number of related things — a record whose shape is known.

`sorted()` orders a collection, and `key=` chooses by what

Python Code | Start Over | Run Code

```
counties = [
    ("Philadelphia", 4052, 1550542),
    ("Allegheny", 2179, 1224825),
    ("Schuylkill", 401, 143786),
    ("Centre", 208, 157795),
    ("Fayette", 325, 123915),
]

by_deaths = sorted(counties, key=lambda record: record[1], reverse=True)

for county, deaths, population in by_deaths[:3]:
    print(f"{county}: {deaths} deaths")
```

`key=` receives a function returning the value to sort on. `lambda` writes that function inline.

Checkpoint: ranking by count answers a different question

Python Code | Start Over | Run Code

```python
counties = [
    ("Philadelphia", 4052, 1550542),
    ("Allegheny", 2179, 1224825),
    ("Schuylkill", 401, 143786),
    ("Centre", 208, 157795),
    ("Fayette", 325, 123915),
]

by_deaths = sorted(counties, key=lambda r: r[1], reverse=True)
by_rate = sorted(counties, key=lambda r: r[1] / r[2] * 100000, reverse=True)

print("Ranked by deaths:")
for county, deaths, population in by_deaths[:3]:
    print(f"   {county}: {deaths} deaths")

print("Ranked by rate:")
for county, deaths, population in by_rate[:3]:
    print(f"   {county}: {deaths / population * 100000:.1f} per 100,000")
```

Counts rank by population; rates rank by burden

The two orderings answer different questions. Ranking Pennsylvania's 67 counties by COVID death **count** returns the five largest counties. Ranking by **rate** returns five rural counties, and **the two lists share no members at all.**

A county appears at the top of a count ranking largely because many people live there.

Objects.

What the dot in `.items()`, `.append()`, and `.strip()` means.

Every value in Python is an object

An object carries **data** and the **operations** defined for it. The dot accesses both.

Python Code | Start Over | Run Code

```
1 county = "Schuylkill"
2 counties = ["Schuylkill", "Centre"]
3 covid_deaths = {"2022-01": 4294}
4
5 print(county.upper())
6 print(counties.count("Centre"))
7 print(covid_deaths.keys())
```

`"text".upper()`, `counties.append()`, and `covid_deaths.items()` are the same construction: **a value, a dot, and an operation that value supports.**

Methods belong to types

`.upper()` exists on strings and not on lists. Calling a method a type does not define raises `AttributeError`, which names both the type and the missing attribute.

A class defines a new kind of object

Python Code | Start Over | Run Code

```python
class County:
    """A Pennsylvania county with COVID-19 mortality data."""

    def __init__(self, name, deaths, population):
        self.name = name
        self.deaths = deaths
        self.population = population

    def rate_per_100k(self):
        """Return deaths per 100,000 residents."""
        return self.deaths / self.population * 100000


schuylkill = County("Schuylkill", 401, 143786)

print(schuylkill.name)
print(f"{schuylkill.rate_per_100k():.1f} per 100,000")
```

1. `__init__` runs when the object is created and stores its data
2. `self` refers to the object being operated on
3. An **attribute** holds data; a **method** performs an operation

Attributes and methods differ by the parentheses

`schuylkill.name` retrieves stored data. `schuylkill.rate_per_100k()` calls a function. Omitting the parentheses returns the function itself rather than its result.

The dot is the same idea everywhere

Python Code | Start Over | Run Code

```python
class County:
    def __init__(self, name, deaths, population):
        self.name = name
        self.deaths = deaths
        self.population = population

    def rate_per_100k(self):
        return self.deaths / self.population * 100000


counties = [
    County("Philadelphia", 4052, 1550542),
    County("Schuylkill", 401, 143786),
    County("Centre", 208, 157795),
]

for county in sorted(counties, key=lambda c: c.rate_per_100k(), reverse=True):
    print(f"{county.name}: {county.rate_per_100k():.1f} per 100,000")
```

What Module 4 introduces

`pandas` supplies an object holding an entire table. `df.head()` and `df.groupby()` use the construction on this slide — a value, a dot, an operation.

The worst month.

Putting the session together to answer the question it opened with - which month was the worst?

Exercise: find the deadliest month

Exercise | Start Over | Show Hint | Run Code

```
covid_deaths = {
    "2021-08": 555, "2021-09": 1248, "2021-10": 2035, "2021-11": 2117,
    "2021-12": 3365, "2022-01": 4294, "2022-02": 1819,
}

worst_month = ""
worst_deaths = 0

for month, deaths in covid_deaths._____():
    if deaths > worst_deaths:
        worst_deaths = deaths
        worst_month = month

print(f"Worst month: {worst_month} with {worst_deaths:,} deaths")
```

January 2022

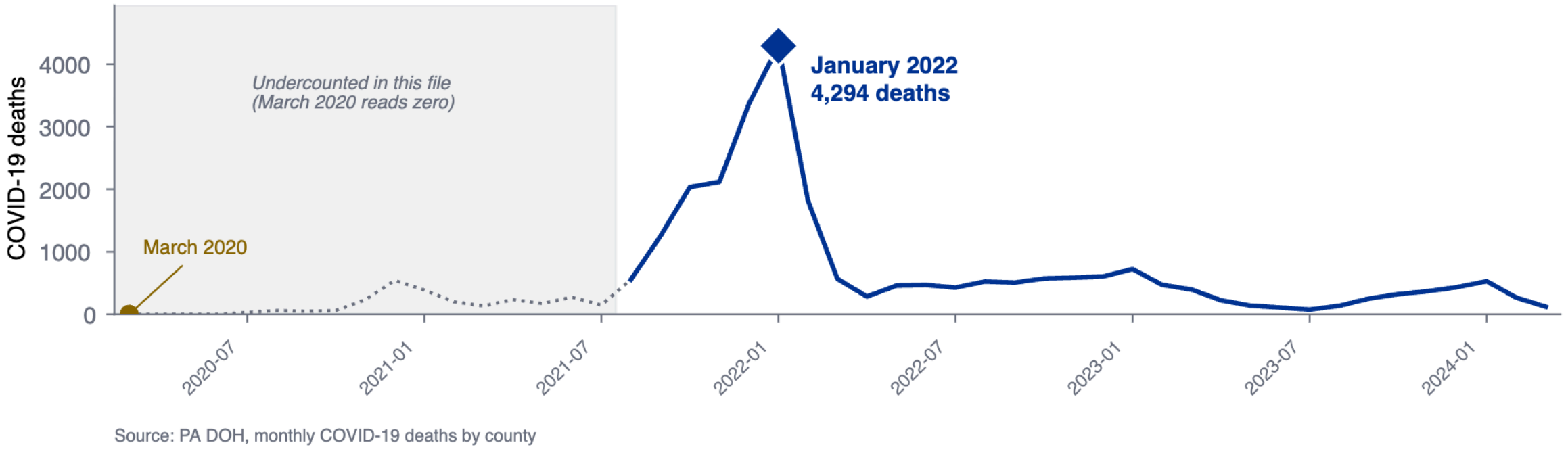


Figure 1. Statewide COVID-19 deaths for each of the 49 months in the file. The series rises through late 2021 to a sharp peak of 4,294 deaths in January 2022, marked by a blue diamond, then falls away and stays low through 2023. Months before August 2021 are shaded and drawn as a dotted line because the file undercounts them — March 2020 reads zero.

4,294 Pennsylvanians died of COVID-19 in January 2022 — the deadliest month of the pandemic in this state.

Not March 2020. Not the first wave.

What the timing means

Challenge: the month it accelerated most

The deadliest month need not be the month deaths **rose** the most. Find the largest rise.

Exercise | Start Over | Show Hint | Run Code

```
def biggest_jump(series):
    previous = None
    best_month = ""
    best_change = 0
    for month, deaths in series.items():
        if previous is not None:
            change = deaths - ______
            if change > best_change:
                best_change = change
                best_month = month
        previous = ______
    return best_month, best_change
month, change = biggest_jump(covid_deaths)
print(f"Largest rise: {month}, up {change:,} deaths")
```

Your turn!

Why is `if previous is not None` needed? And why is the answer not January 2022?

Session outcomes

1. Storing labelled data in a **dictionary**, and retrieving it with `[key]` and `.get()`
2. Iterating with `for`, over a collection and over `range()`
3. Walking key–value pairs with `.items()`
4. Building results with the **accumulator pattern**, initialized outside the loop
5. Repeating with `while`, and exiting with `while True:` plus `break`
6. Defining functions with `def`, `return`, and a docstring
7. Returning several values, and unpacking **tuples**
8. Ordering with `sorted(key=)`, and distinguishing **counts from rates**
9. Reading a small class: `__init__`, `self`, attributes, and methods

> **Preparation for the lab, Tuesday 10/20**
>
> The lab builds county-level dictionaries, writes functions returning several values, and ranks counties two ways.

Module 3 replaces these loops with arithmetic on whole arrays.

That’s all folks!

See you on Tuesday!