

Testing and QA

What tests would you write for this function?

A bakery is using software to fine tune how many pies to bake each day. It doesn't want to run out of pies to sell, but it also doesn't want too many pies left unsold at the end of the day.

Test a function `get_change_in_number_of_pies_to_bake()`. The function:

- Takes the number of pies leftover at the end of the day yesterday
- Returns the change in the number of pies we should bake today

The algorithm for the change in the number of pies to bake is:

- If 0 pies were left over, increase the number of pies we bake by 40
- If 1-20 pies were left over, don't change how many pies we bake
- If more than 20 pies were left over, reduce the number of pies we bake today by: the amount of pies left over yesterday, minus 20

Examples:

```
get_change_in_number_of_pies_to_bake(0)    # 40
get_change_in_number_of_pies_to_bake(9)    # 0
get_change_in_number_of_pies_to_bake(31)   # -11
```

Python ▼

Input and output are both integers.

Answer

For this function, different *ranges* of inputs lead to different behavior. This'll help us choose which inputs to test.

Specifically, we care about these ranges of the number of pies left over:

- Below 0
- 0
- 1 to 20
- Above 20

We should also ask the interviewer if there's an upper bound on the input. Having 50 pies left over might be a normal disappointing day, but something's seriously wrong if we have a *billion* unsold pies, right? Let's say the most pies we can bake in a day is 900, so we'll treat anything over 900 as invalid input.

Ok, so these are our ranges:


- Below 0
- 0
- 1 to 20
- 21 to 900
- Above 900

Identifying these ranges is useful because testing *only* the five numbers -1, 0, 10, 50, and 901 would be more effective than testing *hundreds* of numbers between 21 and 900. To our function, all the numbers between 21 and 900 are effectively equivalent. This technique of identifying ranges is called **equivalence partitioning**.

So is testing any 5 numbers in our 5 ranges enough?

No. It would give us a good idea that the basic logic works, but we might miss off-by-one errors if we're not careful about the *edges* of the ranges.

At a minimum we should test all these inputs at the boundaries of our ranges:



-1, 0, 1, 20, 21, 900, 901

Testing these values is called **boundary value analysis**.

Next, we should ensure we don't accept or return decimals. The input and output were specified as integers (which makes sense—we can't bake a fraction of a pie).

What if we get an invalid input type, like a string? Different languages handle this differently. Java, C#, C++ and C wouldn't let you pass the wrong type of argument, Python and Ruby might give you a `TypeError` (or worse, might not give you an error), and JavaScript will let you get away with almost anything. So the importance of making sure we get an error for invalid input types depends on our language.

Now how would you test *this* function?

We have a function to find the *n*th decimal digit of Pi, and we want to make sure it works.

The function is one-based starting with the first decimal. So the first decimal digit is 1, the second is 4, the third is 1, and so on:

```
3.14159265... # Pi
12345678...  # n
```

Examples:

```
get_nth_digit_of_pi(1)    # 1
get_nth_digit_of_pi(4)    # 5
get_nth_digit_of_pi(71)   # 0
get_nth_digit_of_pi(900)  # 3
get_nth_digit_of_pi(-4)   # ValueError: no nonpositive digits
get_nth_digit_of_pi(0)    # ValueError: no nonpositive digits
```

Python ▼

Answer

We kind of have ranges like the last question—"above zero" and "zero or below." But other than that we don't have any bounds on the input or any changes in behavior based on the input. Sure, 0 and 1 will be important to test, but those aren't enough on their own.

In general for any function that takes numbers as input, it's a good idea to consider testing:

- Negative numbers
- 0
- 1
- 2
- 10, and higher by orders of magnitude
- The highest Integer (like the highest 32 bit integer, or something more specific to your system or language)

This generally covers a basic set of edge cases, potential special cases, base cases, and a reasonable range of values.

The bakery got a new POS system. How would you test checking out with a credit card?

Answer

It won't be enough to see if the system takes a valid credit card and processes a payment. We need to also make sure the system *rejects* invalid cards and gracefully handles cases when things go wrong.

Tests that check if the system works with valid input are called **positive tests**, and tests that check if the system gives the correct behavior for invalid input are called **negative tests**.

Let's get started. We should test a few areas:

Credit card information

From swiping and manual entry, the system should correctly handle:

- **Credit card numbers** that are valid, invalid, an invalid length, and blank.
- **Expiration dates** that are valid, expired, not expired but incorrect, and blank.
- **CVV numbers** that are correct, incorrect, an invalid length, and blank.

Physical things about the card

A card might have an unreadable strip or be damaged. Or it might be swiped incorrectly. If the system has a chip reader, we should cover that too.

Limits and restrictions

Do we have a self-imposed limit on the amount we can charge? Or the amount of times we can run the same card? Do we reject certain credit card companies, like American Express?

Failure

Finally, we should make sure we get safe, expected, reasonable behavior when the system fails. What should happen when the internet or bank communication system goes down? What about after an improper shutdown, like a power outage or if the terminal is suddenly unplugged? And should something happen if the terminal is in the middle of a transaction and there's no activity for an unexpectedly long amount of time?

How many scenarios do we need to test for this function?

`get_cake_price()` determines what to charge for a cake. It has 3 parameters:

```
size          # string, either '1-10 guests' or '11-20 guests'
writing       # string, max 30 characters, blank allowed
pickup_time  # datetime object
```

Python ▼

(writing is what we'll write on the cake.)

Here's how the values are set by the user:

- size is a radio button. 1-10 guests is selected by default.
- writing is a text input field, limited to 30 characters. The field is blank by default.

- `pickup_time` is selected from a calendar, limited to times when the bakery is open. The calendar does not go back in time. Tomorrow at 6:00 pm is selected by default.

And here's how we determine the cost of a cake:

- \$45 for 1 to 10 guests
- \$75 for 11 to 20 guests
- +\$15 for writing *and* same day pickup

Answer

Let's start by looking at the possible values for each parameter.

The size of the cake is easy—it can only be one of two options, '1-10 guests' or '11-20 guests'. The radio button with a default is nice because we know we don't have to worry about invalid or blank input.

But wait—do we know we don't have to worry about invalid or blank input? We got specifications of the field types and defaults in the UI, but what if we have an API and someone directly hits it?

This is a good question to ask your interviewer because it shows an ability to think of things that could go wrong. In this case, the spirit of the question doesn't involve catching edge cases, so we'll assume the arguments are reliably and intentionally restricted, and you can count on valid input.

The writing on the cake can be blank or present. We don't have to worry about it being too long because the input is already limited to 30 characters. We might want to ask our interviewer if we allow *any* 30 characters. What about obscure special characters or offensive words? We'll say there's no restriction—these problems are rare enough and messy (<https://www.washingtonpost.com/news/morning-mix/wp/2018/05/22/proud-mom-orders-summa-cum-laude-cake-online-publix-censors-it-to-summa-laude>) enough that we can just follow up by phone if we're concerned about the message.

For pickup time, looks like we only have 2 values to look for. Times today or times later than today. We know the input is a calendar, has a default, and doesn't allow times when the bakery is closed. So we don't have to worry about blank or invalid input. Although a reasonable question would be to ask about a limit in the *future*—how far in advance can someone order a cake? We'll say there's no limit here.

Ok, let's lay this all out. What are all the combinations of inputs we can have?

Well, we have 2 possibilities for size, 2 possibilities for writing, and 2 possibilities for pickup time. So that's 8 combinations ($2 \times 2 \times 2 = 8$).

Size	1-10	1-10	1-10	1-10	11-20	11-20	11-20	11-20
Writing	Blank	Blank	Present	Present	Blank	Blank	Present	Present
Same day	✓	✗	✓	✗	✓	✗	✓	✗

We know enough to answer the question now, but let's take it a step further and look at the price for each scenario.

Size	1-10	1-10	1-10	1-10	11-20	11-20	11-20	11-20
Writing	Blank	Blank	Present	Present	Blank	Blank	Present	Present
Same day	✓	✗	✓	✗	✓	✗	✓	✗
Price	45	45	60	45	75	75	90	75

This kind of table—showing the possible combination of values for a set of conditions, and identifying the actions we take—is called a **decision table**. This kind of thinking will get even more helpful as the total number of parameters and possible values grows. What if we had different prices for different flavors? What if the inputs could be blank or invalid? Decision tables help us methodically design comprehensive tests.

Sometimes decision tables can also help combine scenarios that have the same action (for example, we could combine the first 2 columns and say "if a cake is for 1-10 guests and writing is blank, it doesn't matter what the pickup time is—we charge \$45"). But this can be dangerous for testing—what if a bug factors in the pickup time when it shouldn't?

So for this this function, we should test 8 scenarios—every possible combination of inputs.

What types of tests should we write?

This question is intentionally vague. It's designed to see if you have an organized way of thinking about test design.

Answer

We should make sure we consider these kinds of tests:

Unit tests and Integration tests. Unit tests cover single functions or small testable units. Integration tests cover how larger components work together. Unit tests can detect low level bugs before they affect other areas of a program, and can help with design and development. Integration tests catch a broader range of bugs because they cover interactions and can be designed around actual use cases. A good test suite usually involves a *mix* of both integration and unit tests.

Regression tests cover bugs after they're caught and fixed. Many teams have a standard that whenever a bug is fixed, a test should be added that fails without the fix and passes with the fix. The idea here is that we found something wrong and we fixed it—let's not break it again

Load and stress testing. Load testing puts a system under a high but realistic load, usually to make sure the system can handle it without any serious impacts on performance. Stress testing looks at what happens when a system is *overwhelmed* beyond what it can handle, usually to make sure the system fails gracefully (say, by giving useful error messages instead of freezing). Load and stress tests are both important for scaling.

Security testing. Software should never expose sensitive data, have unprotected authentication credentials, or carelessly trust user input. **Penetration testing** (or just **pen testing**), for example, tries to harmlessly exploit vulnerabilities in a system so they can be patched. The more sensitive the data you hold, the more important security testing is.

Our program is complex enough that it's infeasible for us to test *every* combination of inputs or *every* path. How should we choose what to test and what not to test?

Answer

One way to start deciding what to test is **risk-based testing**, focusing on what's most likely to have a bad impact. For example, failure in the signup flow might completely halt new business. Or if we charge customers the wrong amount we could make users angry and even get in legal trouble.

We can also break down the **severity and priority** of potential problems. Sometimes there's a tricky balance between the two—for example, are we more worried about a feature that *fails occasionally* for the 80% of users using Chrome or Safari, or a feature that's *totally unusable* to the 5% on Internet Explorer?

A specific way to get reasonable coverage with fewer tests is **combinatorial testing**. Let's say we have an app to regularly check and record our bakery's refrigerator temperatures. We have 2 digital thermometer models (T and TX), 2 app versions (iOS and Android), and 2 connection methods (bluetooth and USB). Similar to a decision table, we can lay out every possible combination. There are 8 configurations for getting refrigerator temperatures:

Test case	Model	Version	Connection
1	T	iOS	Bluetooth
2	T	iOS	USB
3	T	Android	Bluetooth
4	T	Android	USB
5	TX	iOS	Bluetooth
6	TX	iOS	USB
7	TX	Android	Bluetooth
8	TX	Android	USB

But with combinatorial testing, we can reduce this to 4 tests by just testing *pairs*. Instead of asking "Does every model work on every version with every connection?" we can separately ask:

- Do all the models and versions work together?
- Do all the models and connections work together?
- Do all the versions and connections work together?

Test case	Model	Version	Connection
1	T	iOS	Bluetooth
2	T	Android	USB
3	TX	iOS	USB

4	TX	Android	Bluetooth
---	----	---------	-----------

We've reduced our total tests by half, and this approach becomes even more powerful as the number of conditions and the possible values for each condition grows. It's not unrealistic to reduce a quintillion (a billion billion) possible combinations down to hundreds of tests.

If we're worried that just testing *pairs* isn't enough, we can test the interaction of 3 values or any n values. Research shows that the interaction of 6 values is usually enough to find all the bugs in a system, even for complex and dangerous systems like air traffic collision avoidance, medical devices, and space flight.

So combinatorial testing can save us a huge amount of time with no cost or a low cost to effectiveness.

What's next?

Check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now →

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.