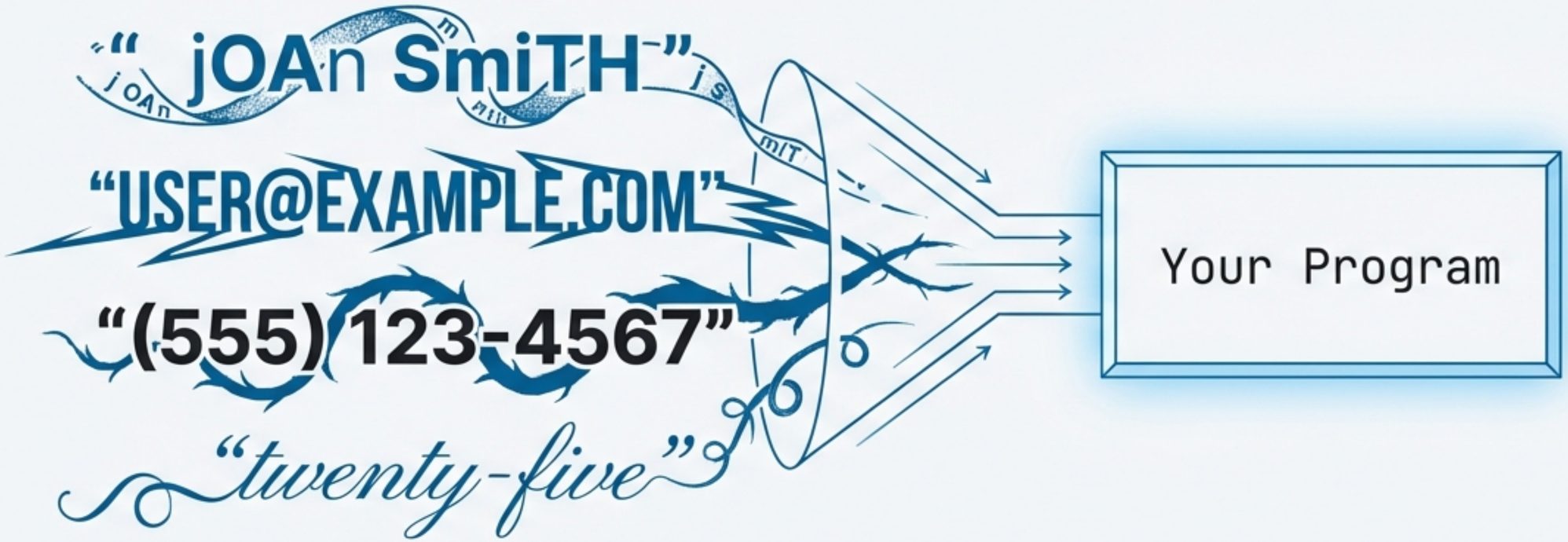
Every Program Starts with Messy Text

User input is unpredictable. It arrives with extra spaces, inconsistent capitalization, and mixed formats. Before you can use data, you must clean and validate it. This is a fundamental skill in software development.



How do we transform this chaos into clean, reliable data?

Our Mission: Build a Contact Card Validator

We will build a Python program that takes messy contact details and transforms them into a clean, **validated, and perfectly formatted contact card.** Each concept you learn will be a new tool to help build this validator.

RAW INPUT

```
name = "   jOAn SmiTH "
   email = "USER@EXAMPLE.COM  "
phone = "(555) 123-4567"
    age = "25"
```

VALIDATED OUTPUT

Name:	Joan Smith [✓ Valid]
Email:	user@example.com [✓ Valid]
Phone:	(555) 123-4567 [✓ Valid]
Age:	25 (Adult) [✓ Valid]

The First Step: Cleaning Names with String Methods

String methods are actions you perform on text. Let's start with two essential methods for cleaning names: `.strip()` removes leading/trailing whitespace, and `.title()` fixes capitalization.

Cleaning a Messy Name

```
# Input with extra spaces and bad capitalization
messy_name = "  jOAn SmITH "

# Chain methods to clean and clean
# 1. .strip() removes whitespace -> "jOAn SmITH"
# 2. .title() capitalizes each word -> "Joan Smith"
cleaned_name = messy_name.strip().title()

print(cleaned_name)
```

Removes whitespace from both ends.

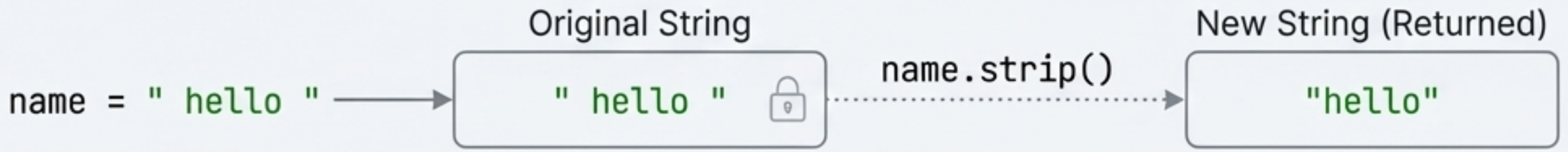
Capitalizes the first letter of each word.

```
Joan Smith
```

Expert Insight: Strings Are Immutable.

A core principle in Python: strings cannot be changed after they are created. Methods like `.strip()` or `.upper()` don't modify the original string; they return a **new, modified string**. You must save this new string to a variable to use it.

What Python Sees



The Correct Pattern

❌ This WON'T work

```
# The result "hello" is created, but not saved.
name = "  hello  "
name.strip()
# print(name) -> "  hello  "
```

✅ This is the correct way

```
# Reassign the variable to the new string.
name = "  hello  "
name = name.strip()
# print(name) -> "hello"
```

Immutability makes strings predictable and safe. Once created, you know a string will never secretly change.

Validating the Email: Normalization and Searching.

For emails, consistency and structure are key. We use `.lower()` to ensure all emails are in lowercase for easy comparison, and `.find()` to check for essential characters like the '@' symbol.

Checking an Email Address

```
email = "  USER@EXAMPLE.COM"

# 1. Clean whitespace and normalize case
cleaned_email = email.strip().lower()  # -> "user@example.com"

# 2. Validate structure: .find() returns the index of a character.
#    If not found, it returns -1.
is_valid = cleaned_email.find("@") >= 0

print(f"Email: {cleaned_email}")
print(f"Contains '@': {is_valid}")
```

A professional pattern to check if a substring exists.

```
Email: user@example.com
Contains '@': True
```

Deconstructing the Phone Number.

Phone numbers come in many formats. Our goal is to extract only the digits. We can chain the `.replace()` method to remove common formatting characters. Then, we use the built-in `len()` function to count the digits and validate the length.

Standardizing a Phone Number

```
phone = "(555) 123-4567"

# Chain .replace() to remove all formatting
cleaned_phone = phone.replace("(", "") \
                     .replace(")", "") \
                     .replace(" ", "") \
                     .replace("-", "")  # -> "5551234567"

# Validate the length
is_valid_length = len(cleaned_phone) == 10

print(f"Digits: {cleaned_phone}")
print(f"Valid Length: {is_valid_length}")
```

```
Digits: 5551234567
Valid Length: True
```

`len()` is a built-in function, `len(text)`, while `.replace()` is a string method, `text.replace()`. Notice the different syntax.

Our Validator So Far: Cleaning and Structuring.

Let's assemble the pieces we've built. Our program can now take raw name, email, and phone strings, apply a series of cleaning and validation methods, and prepare them for the next steps.

```
# --- Contact Card Validator v1.0 ---

# 1. Input Data
name_input = "  jOAn SmITH "
email_input = "USER@EXAMPLE.COM  "
phone_input = "(555) 123-4567"

# 2. Process Name
cleaned_name = name_input.strip().title()  Clean & Capitalize
name_valid = len(cleaned_name) > 0

# 3. Process Email
cleaned_email = email_input.strip().lower()  Clean & Normalize
email_valid = cleaned_email.find("@") > 0  # Simple check

# 4. Process Phone
cleaned_phone = phone_input.replace("(", "").replace(")", "") \  Remove Formatting
                           .replace(" ", "").replace("-", "")
phone_valid = len(cleaned_phone) == 10 and cleaned_phone.isdigit()  Validate Content

# --- (Validation results would be checked here) ---
```

The Next Challenge: Age is Just a String.

Our validator needs to check if a person is an adult (e.g., age >= 18). But all user input, even "25", is a string. You cannot perform mathematical comparisons on strings.

JetBrains Mono

Why This Fails

```
age_string = "25"

# This will raise a TypeError!
# You can't use ">=" to compare a string and an integer.
is_adult = age_string >= 18
```

```
TypeError: '>=' not supported between instances of 'str' and 'int'
```

How do we safely convert a string like "25" into a number we can actually use for calculations?

The Solution: Type Casting with `int()`.

Type casting is explicitly converting data from one type to another. The `int()` function can parse a string and return an integer, but it will fail if the string doesn't look exactly like an integer.

Success Case ✅

```
age_string = "25"
age_number = int(age_string)

print(age_number + 5) # Now it's a number!
```

```
30
```

Failure Case ❌

```
invalid_age_string = "twenty-five"

# This raises a ValueError!
age_number = int(invalid_age_string)
```

```
ValueError: invalid literal for
int() with base 10: 'twenty-five'
```

Expert Insight: The Validation-First Mindset.

Errors are information. A `ValueError` is Python protecting you from bad data. Instead of crashing, professionals validate input *before* attempting to convert it. This habit will save you hours of debugging.

JetBrains Mono

How to Prevent Crashes

```
age_input = "25" # Try with "25", then "twenty-five"

age_number = None # Default value
if age_input.strip().isdigit():
    # Only convert if it's safe!
    age_number = int(age_input)
    print(f"Age successfully converted: {age_number}")
else:
    print(f"'{age_input}' is not a valid number.")
```

The `.isdigit()` method checks if a string contains only numbers. It's your safety check before calling `int()`.

Clean → Validate → Convert → Use. This is the workflow for handling any user input.

Presenting the Result: Formatting with F-Strings.

F-strings (formatted string literals) are Python's best tool for creating dynamic text. Prefix a string with `f` and place variables or any valid Python expression inside curly braces `{}`.

Building the Formatted Card

```
# Assume these variables are cleaned and validated
name = "Joan Smith"
phone = "5551234567"
age = 25

# Format the phone number using slicing inside the f-string
formatted_phone = f"({phone[0:3]}) {phone[3:6]}-{phone[6:10]}"

# Create the final output
contact_card = f"""
--- CONTACT CARD ---
Name: {name.upper()}
Phone: {formatted_phone}
Age: {age} (Is Adult: {age >= 18})
--------------------
"""
print(contact_card)
```

```
--- CONTACT CARD ---
Name: JOAN SMITH
Phone: (555) 123-4567
Age: 25 (Is Adult: True)
--------------------
```

The Complete Data Wrangler.

Here is our finished validator. It combines string methods for cleaning, `len()` and `.find()` for validation, type casting for numerical logic, and f-strings for polished output. You have successfully tamed the chaos.

```
# --- Contact Card Validator v2.0 ---

# 1. Input Data
name_input = "  jOAn SmITH "
email_input = "USER@EXAMPLE.COM  "
phone_input = "(555) 123-4567"
age_input = "  25  "

# 2. Clean and Validate Name, Email, Phone
cleaned_name = name_input.strip().title()
name_valid = len(cleaned_name) > 0

cleaned_email = email_input.strip().lower()
email_valid = cleaned_email.find("@") > 0 and "." in cleaned_email

digits = "".join(c for c in phone_input if c.isdigit())
phone_valid = len(digits) == 10

# 3. Clean, Validate, and Convert Age
cleaned_age_str = age_input.str1p()
age_number = None
if cleaned_age_str.isdigit():
    age_number = int(cleaned_age_str)

# 4. Final Check and Formatted Output
if name_valid and email_valid and phone_valid and (age_number is not None):
    print("--- VALID CONTACT ---")
    # (f-string formatting for the final card would be here)
else:
    print("--- INVALID CONTACT ---")
    # (Print error messages for each invalid field)
```

Input Chaos

```
" jOAn SmITH "
                "USER@EXAMPLE.COM "
"(555) 123-4567"     " 25 "
```

Validated Order

```
--- CONTACT CARD ---
Name: JOAN SMITH
Phone: (555) 123-4567
Age: 25 (Is Adult: True)
--------------------
```

Putting Our Validator to the Test

A robust program handles not just perfect data, but also messy, invalid, and edge-case inputs gracefully. Here's how our validator performs in real-world scenarios.

Test Scenario	Input	Program Output
Messy but Valid	`name='  bob jones  '` `age=' 30 '`	`--- VALID CONTACT --- ` `Name: Bob Jones...`
Invalid Email	`email='bob@invalid' (no .com`)`	`--- INVALID CONTACT --- ` `Email format is invalid.`
Invalid Age	`age='thirty'`	`--- INVALID CONTACT --- ` `<b>Age must be a number.`
Invalid Phone	`phone='123-4567'`	`--- INVALID CONTACT ---` ` Phone must be 10 digits.`
Edge Case: Empty	`name='  '`	`--- INVALID CONTACT ---` ` Name cannot be empty.`

How Each Skill Solved the Problem

The validator wasn't built with one tool, but a combination of skills, each with a specific purpose.

String Methods

Our tools for **Cleaning & Structuring.** They normalized case, removed junk characters, and checked for patterns.

`.strip`, `.title`, `.lower`, `.replace`, `.find`

Fundamentals

Our tools for **Counting & Accessing.** They validated length and allowed us to extract parts of a string for reformatting.

`len`, `[] indexing`

Type Casting

Our tools for **Conversion & Safety.** They transformed text into numbers for logic and prevented our program from crashing on bad

`int`, `.isdigit`

F-Strings

Our tool for **Presentation.** They allowed us to create a clean, dynamic, and readable final output.

`f"{}"`

You Are Now a Data Wrangler

You've completed the quest. You didn't just learn about strings; you learned how to engineer data quality. The skills you've mastered are used every day to build registration forms, process payments, and manage user data in professional applications.

The Data Wrangler's Toolkit

Cleaning Kit

```
.strip()
.lower()
.upper()
.title()
.replace()
```

Validation Kit

```
.find()
.isdigit()
len()
isinstance()
```

Conversion Kit

```
int()
float()
str()
```

Formatting Kit

```
F-Strings
```

You have the tools. Go build something reliable.