Building Calculation Specifications: A Complete Guide

[Your Name]

5 February 2026

Table of contents

For Pension Analysts Moving from Word/Excel to Jupyter Notebooks

Field	Value
Version	1.0
Audience	Pension Analysts
Prerequisites	None (we'll teach you everything)

0.1 Why Change How We Write Specs?

0.1.1 The Problem with Word Documents

We've all been there:

- You write a 20-page spec with formulas
- A developer implements it
- Testing finds a bug… but is it in the spec or the code?
- Nobody can tell because the spec formulas were never actually *run*
- Hours of meetings to trace through the logic manually

0.1.2 The Problem with Excel

- Formulas are hidden in cells — hard to document
- Cell references like `=B7*C12` are meaningless without context
- Version control is painful
- One accidental edit breaks everything

0.1.3 The Solution: Executable Specifications

What if your spec document could:

1. **Run the actual calculation** with test data
2. **Show each step** so everyone can follow along
3. **Prove it works** before a developer ever sees it
4. **Be the single source of truth** — one document, not spec + code + tests

That's what this approach gives you.

---

1 Part 1: Python Basics (15 minutes to learn)

You don't need to become a programmer. You need to learn about **5 things**.

1.1 1.1 Variables: Named Values

Instead of cell references, we use meaningful names.

```
# This is a comment - it explains the code but doesn't run

# Variables are just named storage boxes
pension_at_leaving = 15000.00
gmp_at_leaving = 3500.00
years_of_service = 25
member_name = "John Smith"      # Text goes in quotes
is_married = True               # True or False (boolean)

# Print them to see the values
print(pension_at_leaving)
print(member_name)
```

```
15000.0
John Smith
```

Compare to Excel:

Excel	Python
Cell B2 contains 15000	`pension_at_leaving = 15000`
You have to remember B2 means pension	The name tells you what it is
`=B2-B3`	`excess = pension_at_leaving - gmp_at_leaving`

Which is clearer?

1.2 1.2 Basic Arithmetic: Exactly Like Excel

```
# Addition, subtraction, multiplication, division
a = 100
b = 25

print(f"Addition:       {a} + {b} = {a + b}")
print(f"Subtraction:    {a} - {b} = {a - b}")
print(f"Multiplication: {a} * {b} = {a * b}")
print(f"Division:       {a} / {b} = {a / b}")

# Powers (like Excel's POWER function or ^)
rate = 0.03
years = 5
factor = (1 + rate) ** years    # ** means "to the power of"
print(f"")
print(f"Compound factor: (1 + {rate}) ^ {years} = {factor:.4f}")
```

```
Addition:       100 + 25 = 125
Subtraction:    100 - 25 = 75
Multiplication: 100 * 25 = 2500
Division:       100 / 25 = 4.0

Compound factor: (1 + 0.03) ^ 5 = 1.1593
```

1.3 1.3 Conditionals: Making Decisions (IF statements)

This is like Excel's `IF()` function.

```
# Simple if/else - like Excel's =IF(condition, value_if_true, value_if_false)

gender = "M"

if gender == "M":           # Note: == means "equals" (comparison)
    retirement_age = 65
else:
    retirement_age = 60

print(f"Gender: {gender}")
print(f"Normal Retirement Age: {retirement_age}")
```

```
Gender: M
Normal Retirement Age: 65
```

```
# Multiple conditions - like nested IFs in Excel

years_of_service = 12

if years_of_service < 2:
    benefit_type = "Refund of contributions"
elif years_of_service < 5:    # "elif" means "else if"
    benefit_type = "Short service benefit"
else:
    benefit_type = "Full deferred pension"

print(f"Years of service: {years_of_service}")
print(f"Benefit type: {benefit_type}")
```

```
Years of service: 12
Benefit type: Full deferred pension
```

```
# Combining conditions with 'and' / 'or'

age = 58
years_of_service = 30

# Rule 85: age + service >= 85
if age + years_of_service >= 85:
    print("Meets Rule of 85 - unreduced early retirement available")
else:
    print("Does not meet Rule of 85")

# Multiple conditions
if age >= 55 and years_of_service >= 2:
    print("Eligible for early retirement")
```

```
Meets Rule of 85 - unreduced early retirement available
Eligible for early retirement
```

1.4 1.4 Printing Results: Showing Your Work

Use `print()` with f-strings to display results clearly.

```python
pension = 15432.50
factor = 0.82567

# Basic print
print("Basic:", pension)

# f-string (formatted string) - much nicer
print(f"Pension is £{pension}")

# With formatting
print(f"Pension: £{pension:,.2f}")     # Commas and 2 decimal places
print(f"Factor:  {factor:.4f}")        # 4 decimal places
print(f"Factor:  {factor:.2%}")        # As percentage
```

```
Basic: 15432.5
Pension is £15432.5
Pension: £15,432.50
Factor:  0.8257
Factor:  82.57%
```

Format codes explained:

Code	Meaning	Example	Output
`{x}`	As-is	`f"{15000}"`	15000
`{x:,}`	With commas	`f"{15000:,}"`	15,000
`{x:.2f}`	2 decimal places	`f"{15000:.2f}"`	15000.00
`{x:,.2f}`	Both	`f"{15000:,.2f}"`	15,000.00
`{x:.4f}`	4 decimal places	`f"{0.8257:.4f}"`	0.8257
`{x:.2%}`	As percentage	`f"{0.035:.2%}"`	3.50%

1.5 1.5 That's It!

With just these concepts, you can write pension calculations:

- **Variables** — named values
- **Arithmetic** — +, -, *, /, **
- **Conditionals** — if/elif/else
- **Print** — show results

Everything else is just combinations of these basics.

---

2 Part 2: Working with Factor Tables

This is where Python really shines. No more VLOOKUP nightmares.

2.1 2.1 Simple Lookup Tables (Dictionaries)

A dictionary is like a two-column lookup table.

```python
# Early retirement factors by years early
# Format: { key: value, key: value, ... }

erf_male = {
```

```
    0: 1.000,
    1: 0.940,
    2: 0.882,
    3: 0.826,
    4: 0.772,
    5: 0.720,
}

# Look up a value
years_early = 3
factor = erf_male[years_early]

print(f"Years early: {years_early}")
print(f"ERF: {factor}")
```

```
Years early: 3
ERF: 0.826
```

```
# Safe lookup with a default value (if key might not exist)
years_early = 7  # Not in our table!

factor = erf_male.get(years_early, 0.650)  # Returns 0.650 if not found
print(f"Factor for {years_early} years early: {factor}")
```

```
Factor for 7 years early: 0.65
```

2.2 2.2 Two-Dimensional Lookups (Nested Dictionaries)

What if factors depend on TWO things, like age AND gender?

```
# Commutation factors by age and gender
# Outer key = gender, inner key = age

commutation_factors = {
    "M": {
        60: 14.50,
        61: 14.10,
        62: 13.70,
        63: 13.30,
        64: 12.90,
        65: 12.50,
    },
    "F": {
        60: 15.20,
        61: 14.80,
        62: 14.40,
        63: 14.00,
        64: 13.60,
        65: 13.20,
    }
}

# Look up: gender first, then age
gender = "F"
age = 62
```

```
factor = commutation_factors[gender][age]
print(f"Commutation factor for {gender} age {age}: {factor}")
```

```
Commutation factor for F age 62: 14.4
```

2.3 2.3 Large Factor Tables: Keep Them Readable

For big tables, format them nicely in the code.

```
# Transfer value factors - age vs years to NRA
# Laid out like a table for easy reading and checking

tv_factors = {
    #  Age:  {Years to NRA: factor}
    50: {15: 8.50, 14: 8.80, 13: 9.10, 12: 9.40, 11: 9.70, 10: 10.00},
    51: {14: 8.90, 13: 9.20, 12: 9.50, 11: 9.80, 10: 10.10,  9: 10.40},
    52: {13: 9.30, 12: 9.60, 11: 9.90, 10: 10.20,  9: 10.50,  8: 10.80},
    53: {12: 9.70, 11: 10.00, 10: 10.30, 9: 10.60,  8: 10.90,  7: 11.20},
    54: {11: 10.10, 10: 10.40, 9: 10.70, 8: 11.00,  7: 11.30,  6: 11.60},
    55: {10: 10.50,  9: 10.80, 8: 11.10, 7: 11.40,  6: 11.70,  5: 12.00},
}

# Lookup
age = 53
years_to_nra = 10
factor = tv_factors[age][years_to_nra]

print(f"TV factor for age {age}, {years_to_nra} years to NRA: {factor}")
```

```
TV factor for age 53, 10 years to NRA: 10.3
```

2.4 2.4 Factor Tables as Markdown + Code

Best practice: Show the table in Markdown for documentation, then replicate in code for calculation.

Early Retirement Factors

Years Early	Male	Female
0	1.000	1.000
1	0.940	0.945
2	0.882	0.891
3	0.826	0.839
4	0.772	0.789
5	0.720	0.741

```
# Same table in code (copy from above)
erf = {
    "M": {0: 1.000, 1: 0.940, 2: 0.882, 3: 0.826, 4: 0.772, 5: 0.720},
    "F": {0: 1.000, 1: 0.945, 2: 0.891, 3: 0.839, 4: 0.789, 5: 0.741},
}

# Use it
gender = "M"
```

```
years_early = 3
factor = erf[gender][years_early]

print(f"ERF for {gender}, {years_early} years early: {factor}")
```

```
ERF for M, 3 years early: 0.826
```

2.5 2.5 Very Large Factor Sets: External Files

If you have hundreds of factors (e.g., mortality tables), you can load from a CSV file.

```
import csv

# Load factors from CSV
factors = {}
with open('mortality_factors.csv') as f:
    reader = csv.DictReader(f)
    for row in reader:
        age = int(row['age'])
        factors[age] = float(row['qx'])

# Now use it
print(factors[65])  # Mortality rate at age 65
```

But for most pension specs, inline dictionaries are clearer and more self-contained.

---

3 Part 3: Structuring Complex Calculations

How to break down complicated benefit structures.

3.1 3.1 The Golden Rule: One Step Per Cell

Don't try to do everything at once. Each calculation step gets:

1. A **Markdown cell** explaining what we're doing and why
2. A **Code cell** that does the calculation and shows the result

This makes it easy to: - Follow the logic - Debug problems - Verify against the rules

3.2 3.2 Example: Multi-Tranche Calculation

Member has service in three different benefit structures:

Tranche	Period	Accrual	Revaluation
Pre-1997	Before 6/4/1997	1/80th	Fixed 3.5%
1997-2008	6/4/1997 to 5/4/2008	1/60th	CPI capped 5%
Post-2008	After 5/4/2008	1/60th	CPI capped 2.5%

```
# Member data
final_salary = 50000.00
service_pre_1997 = 8
service_1997_2008 = 11
service_post_2008 = 6
```

```
years_of_deferment = 5
cpi_rate = 0.025  # Assumed average
```

3.2.1 Step 1: Calculate pension for each tranche at date of leaving

```
# Tranche 1: Pre-1997 (1/80th accrual)
pension_t1 = final_salary * (1/80) * service_pre_1997

# Tranche 2: 1997-2008 (1/60th accrual)
pension_t2 = final_salary * (1/60) * service_1997_2008

# Tranche 3: Post-2008 (1/60th accrual)
pension_t3 = final_salary * (1/60) * service_post_2008

total_at_leaving = pension_t1 + pension_t2 + pension_t3

print("PENSION AT LEAVING (by tranche):")
print(f"  Pre-1997 (1/80th):    £{pension_t1:>10,.2f}")
print(f"  1997-2008 (1/60th):   £{pension_t2:>10,.2f}")
print(f"  Post-2008 (1/60th):   £{pension_t3:>10,.2f}")
print(f"                        -----------")
print(f"  TOTAL:                £{total_at_leaving:>10,.2f}")
```

```
PENSION AT LEAVING (by tranche):
  Pre-1997 (1/80th):    £  5,000.00
  1997-2008 (1/60th):   £  9,166.67
  Post-2008 (1/60th):   £  5,000.00
                        -----------
  TOTAL:                £ 19,166.67
```

3.2.2 Step 2: Apply revaluation (different rates for each tranche)

```
# Revaluation rates
reval_t1 = 0.035                          # Fixed 3.5%
reval_t2 = min(cpi_rate, 0.05)            # CPI capped at 5%
reval_t3 = min(cpi_rate, 0.025)           # CPI capped at 2.5%

# Apply compound revaluation
pension_t1_reval = pension_t1 * (1 + reval_t1) ** years_of_deferment
pension_t2_reval = pension_t2 * (1 + reval_t2) ** years_of_deferment
pension_t3_reval = pension_t3 * (1 + reval_t3) ** years_of_deferment

total_at_retirement = pension_t1_reval + pension_t2_reval + pension_t3_reval

print("PENSION AT RETIREMENT (after revaluation):")
print(f"  Pre-1997 @ {reval_t1:.1%}:     £{pension_t1_reval:>10,.2f}")
print(f"  1997-2008 @ {reval_t2:.1%}:    £{pension_t2_reval:>10,.2f}")
print(f"  Post-2008 @ {reval_t3:.1%}:    £{pension_t3_reval:>10,.2f}")
print(f"                           -----------")
print(f"  TOTAL:                   £{total_at_retirement:>10,.2f}")
```

```
PENSION AT RETIREMENT (after revaluation):
  Pre-1997 @ 3.5%:     £  5,938.43
  1997-2008 @ 2.5%:    £ 10,371.24
```

```
  Post-2008 @ 2.5%:     £  5,657.04
                          -----------
  TOTAL:                 £ 21,966.71
```

3.3 3.3 Multiple Membership Categories

What if a scheme has different categories (e.g., Officers vs Staff)?

Option A: Separate specs — One notebook per category (recommended if rules are very different)

Option B: Parameterised spec — One notebook that handles all categories

```
# Option B: Parameterised approach

# Member category - CHANGE THIS to test different categories
category = "Officer"  # "Officer" or "Staff"

# Category-specific parameters
if category == "Officer":
    accrual_rate = 1/45
    nra = 60
    spouse_fraction = 0.5
elif category == "Staff":
    accrual_rate = 1/60
    nra = 65
    spouse_fraction = 0.333
else:
    raise ValueError(f"Unknown category: {category}")

print(f"Category: {category}")
print(f"Accrual rate: 1/{int(1/accrual_rate)}")
print(f"Normal retirement age: {nra}")
print(f"Spouse's fraction: {spouse_fraction:.1%}")
```

```
Category: Officer
Accrual rate: 1/45
Normal retirement age: 60
Spouse's fraction: 50.0%
```

---

4 Part 4: Addressing Your Concerns

Common questions and honest answers.

4.1 Q1: "What if the scheme is really complex?"

Answer: Complex schemes are *exactly* where this approach shines.

A complex Word spec is hard to follow. A complex *executable* spec lets you:

- **Test edge cases** by changing inputs and re-running
- **Verify each step** independently
- **Prove it works** with real examples before handoff

If the scheme is so complex you're worried about the spec, imagine being the developer who has to implement it from a 50-page Word document they can't test!

Complexity is a reason TO use this approach, not to avoid it.

4.2 Q2: "What about lots of membership categories?"

Options:

1. **Separate notebooks** — If rules are fundamentally different, separate specs are clearer
2. **Parameterised notebook** — If rules are similar with different parameters, use one spec with a category input
3. **Hybrid** — Common calculations in a base spec, category-specific overrides in separate sections

The key insight: you'd have this complexity in Word too. Here, at least you can test each category.

4.3 Q3: "What about large factor tables?"

Options by size:

Size	Approach
Small (<20 values)	Inline dictionary in code
Medium (20-100 values)	Formatted dictionary, laid out like a table
Large (100+ values)	Load from CSV file alongside the notebook

For very large tables: Put the CSV file in the same folder as the notebook. Document it in Markdown. Load it in code. The factors are still version-controlled and transparent.

Honestly though, most pension factor tables are small enough to put directly in the code, and that's often better because everything is in one place.

4.4 Q4: "Isn't this coding it twice?"

This is the most important objection, so let's address it thoroughly.

4.4.1 The Traditional Process

1. Analyst writes spec in Word (hours/days)
2. Developer reads spec, asks questions (meetings)
3. Developer writes code (days/weeks)
4. QA tests and finds bugs (more meetings: "Is this spec or code?")
5. Back and forth until it works (weeks)

4.4.2 The Executable Spec Process

1. Analyst writes executable spec (similar time to Word, maybe slightly more)
2. Analyst tests it themselves — bugs found before handoff
3. Developer reads spec that *already works* with examples
4. Developer translates to production code (easier — logic is proven)
5. QA compares results to spec output (much faster)

4.4.3 Why It's Not "Twice"

- The spec Python is **simple arithmetic** — not production code
- No error handling, no edge cases, no integration, no performance concerns
- The developer still needs to write proper code with all those things
- But they're translating **working logic**, not interpreting ambiguous prose

4.4.4 The Real Comparison

Task	Word Spec	Executable Spec
Writing formulas	Same effort	Same effort
Verifying correctness	Manual checking	Run and see
Finding errors	After development	Before handoff
Ambiguity	High	Low (code is precise)
Developer questions	Many	Few
Testing baseline	None	Ready-made

4.5 Q5: "I'm not a programmer"

You don't need to be. Look at what you've learned in this document:

- Variables: `pension = 15000`
- Arithmetic: `+`, `-`, `*`, `/`, `**`
- Conditions: `if/else`
- Tables: `factors = {key: value}`
- Output: `print(f"Result: {x}")`

That's it. That's 90% of what you need.

Compare to Excel: you already know formulas, cell references, IF statements, VLOOKUP. This is the same concepts with clearer syntax.

4.6 Q6: "What if I make a mistake in the Python?"

Then you get an error message, and you fix it.

Compare to Word: if you make a mistake in a formula, nobody knows until development or testing — weeks later.

Errors caught immediately are cheap. Errors caught in QA are expensive.

4.7 Q7: "Our actuaries/managers won't read code"

They don't have to. The notebook produces:

- **PDF output** with all the documentation, formulas, AND calculated results
- **HTML output** for sharing digitally
- **The code cells show the formulas** in almost-English: `total = gmp + excess`

Most reviewers will read the Markdown documentation and check the calculated results. That's fine — they're reviewing the *logic and output*, which is what matters.

Anyone who wants to check the implementation can look at the code. It's there, it's transparent.

---

5 Part 5: Getting Started

5.1 Your First Spec: The Process

1. **Copy a template** — Start with the closest match to your calculation
2. **Update the header** — Scheme name, your name, date
3. **List your inputs** — What member data do you need?
4. **Define parameters** — Accrual rates, factors, etc.

5. **Write step by step** — Markdown explanation, then code, then output
6. **Test with real examples** — Use actual member scenarios
7. **Document edge cases** — What special situations exist?

5.2 Tips for Success

- **Start simple** — Get the basic calculation working first
- **Test often** — Run cells as you write them
- **Use real numbers** — Test with actual member examples
- **Show your work** — Print intermediate results
- **Ask for help** — Python errors are usually easy to fix

5.3 Resources

- `QUICK_START.md` — Setup and keyboard shortcuts
- `active_to_retirement_spec.ipynb` — Full template example
- `deferred_retirement_spec.ipynb` — Another template example

---

6 Summary: The Case for Executable Specs

Aspect	Word Spec	Executable Spec
Formulas	Static text	Running code
Testing	Manual review	Click Run All
Errors found	In development/QA	While writing
Ambiguity	Prose is ambiguous	Code is precise
Version control	Track changes mess	Clean Git diffs
Examples	Separate spreadsheet	Embedded and live
Developer handoff	Lots of questions	Logic already proven
QA baseline	None	Spec output = expected results

The goal isn't to make analysts into programmers.

The goal is to make specifications that actually work before anyone tries to implement them.

---

7 Appendix: Quick Reference

7.1 Python Operators

Operator	Meaning	Example
`+`	Add	`5 + 3` → `8`
`-`	Subtract	`5 - 3` → `2`
`*`	Multiply	`5 * 3` → `15`
`/`	Divide	`5 / 2` → `2.5`
`**`	Power	`2 ** 3` → `8`
`==`	Equals	`x == 5`
`!=`	Not equals	`x != 5`
`< >`	Less/greater	`x < 5`
`<= >=`	Less/greater or equal	`x <= 5`

Operator	Meaning	Example
`and`	Both true	`x > 0 and x < 10`
`or`	Either true	`x < 0 or x > 10`

7.2 Common Pension Calculations

```
# Final salary pension
pension = salary * accrual_rate * years

# Revaluation
revalued = original * (1 + rate) ** years

# Early retirement reduction
reduced = pension * erf[years_early]

# Commutation
lump_sum = pension_given_up * commutation_factor

# Split GMP and excess
excess = total_pension - gmp
```

7.3 Dictionary (Lookup Table) Patterns

```
# Simple lookup
factors = {1: 0.94, 2: 0.88, 3: 0.83}
result = factors[key]

# With default
result = factors.get(key, default_value)

# Two-dimensional
factors = {"M": {60: 1.0}, "F": {60: 1.0}}
result = factors[gender][age]
```