

Functions

The big picture

- **Functions**

- example of a bigger idea: modular design avoids redundancy — write the logic once, reuse it everywhere; keeps code organised — each piece has one clear job, easy to find; saves you rewriting functionality every time requirements change

- **Other tools for structuring code**

- modules/packages — functions grouped into reusable files;
- libraries — other people's functions, ready to import
- classes — a way to create your own data types as objects
- methods functions that belong to an object
- design patterns, larger software architectures etc



Overview

- **Warm-up**
 - What is a function and why bother
- **Parameters and arguments**
 - Positional v keyword, defaults, `*args`
- **Return values**
 - Don't `print()`!
- **Write good functions**
 - Tips towards clean code



Basics

- Creating user-defined functions

Recap

- A named, reusable block of code
- Does one job
- You give it a name
- Can take input
- Can hand back a result

```
1  def square(n):  
2      ... return n * n  
3  
4  
5  def main():  
6      ... result = square(5)  
7      ... print(result) ... # 25
```

Why bother?

- Duplicates, untidy

```
1  def main():
2      ... temps_c = [0, 20, 37, 50, 60, -15]
3
4      ... print(temps_c[0] * 9 / 5 + 32)
5      ... print(temps_c[2] * 9 / 5 + 32)
6      ... print(temps_c[3] * 9 / 5 + 32)
7      ... print(temps_c[4] * 9 / 5 + 32)
8      ... print(temps_c[5] * 9 / 5 + 32)
9
10
11  main()
12
```

- Minimal, clean

```
1  def to_fahrenheit(c):
2      ... return c * 9 / 5 + 32
3
4
5  def main():
6      ... temps_c = [0, 20, 37, 50, 60, -15]
7      ... for c in temps_c:
8          ... print(to_fahrenheit(c))
9
10
11  main()
12
```

Why bother?

- **Functions:**

- hide implementation
- name behaviour
- make code more readable

- **Why not bother?**

- this is the question to ask!

```
1 def main():
2     x = 0
3     y = 0
4
5     y -= 1
6     print("Current position:", x, y)
7
8     x += 1
9     print("Current position:", x, y)
10
11    y += 1
12    print("Current position:")
13
14    x -= 1
15    print("Current position:")
16
17
18 main()
19
```

```
21 def main():
22     x = 0
23     y = 0
24
25     x, y = move_north(x, y)
26     print("Current position:", x, y)
27
28     x, y = move_east(x, y)
29     print("Current position:", x, y)
30
31     x, y = move_south(x, y)
32     print("Current position:", x, y)
33
34     x, y = move_west(x, y)
35     print("Current position:", x, y)
36
37
38 main()
```

Parameters & Arguments

- Proper use of
- Defaults
- `*args`

Positional v keyword

- **Positional arguments**
 - matched to parameters by order
- **Keyword arguments**
 - matched by name
 - order stops mattering
 - self-documenting
 - clearer at a glance
- **Mix styles**
 - every positional argument has to come before any keyword argument

```
1  def greet(name, greeting):
2      ... print(greeting, name)
3
4
5  def main():
6      ... # positional
7      ... greet("Richard", "Hi")
8
9      ... # keyword -- order doesn't matter
10     ... greet(greeting="Hi", name="Richard")
11
12
13  main()
```

Default parameters

- **Fallback value**
 - if not specified as argument
- **Optional when calling**
 - leave it out and the default is used
- **Override when needed**
 - pass a value to replace the default
- **Ordering rule**
 - parameters with defaults must come after those defined without

```
1  def greet(name, greeting="Hello"):
2      ... print(greeting, name)
3
4
5  def main():
6      ... greet("Richard") ... # Hello Richard
7      ... greet("Richard", "Hi") ... # Hi Richard
8
9
10 main()
```

Example errors

- No greeting, is required!

```
1 def greet(name, greeting):
2     ... print(greeting, name)
3
4
5 def main():
6     ... greet("Rich") # crash! TypeError:
7
8
9 main()
10
```

- Wrong order!

```
1 def greet(name, greeting):
2     ... print(greeting, name)
3
4
5 def main():
6     ... greet("Rich", greeting="Hi")
7
8     ... greet(greeting="Hi", "Rich") # crash!
9
10
11 main()
```

Parameter `*args`

- **Accept any number of arguments**
 - without naming each one in advance
- **Tells Python**
 - 'collect the extra arguments here'
- **The `*args` not a keyword**
 - `*values`, `*numbers`
 - convention though, instantly recognisable
- **`*args` bundled as a tuple**
 - useful when you don't know how many values are coming

```
1  def total(*numbers):
2      ... return sum(numbers)
3
4
5  def main():
6      ... print(total(1, 2, 3))
7      ... print(total(10, 20))
8      ... print(total(1, 2, 3, 1, 5, 7, 9))
9
10
11  main()
12
```

Return values

Return v print, return `None`

Multiple return values

Classic mistake

- No!

```
1  def add(a, b):
2      ... print(a + b)
3
4
5  def main():
6      ... result = add(2, 3)
7
8      ... # None -- add() never returned anything
9      ... print(result)
10
11
12  main()
```

- Yes!

```
1  def add(a, b):
2      ... return a + b
3
4
5  def main():
6      ... result = add(2, 3)
7      ... print(result) # 5
8
9
10  main()
```

Return multiple values

- A `return a, b` doesn't return two things
 - packs them into a single tuple behind the scenes
- **Unpack that tuple into separate names on the call side:**
 - `lo, hi = min_max(numbers)`
 - Unpack into same number of values as number returned
- **Leave it packed**
 - index into it
 - loop over it, like any other tuple

```
1  def min_max(numbers):
2      ... return min(numbers), max(numbers)
3
4
5  def main():
6      ... lo, hi = min_max([4, 1, 9, 2])
7      ... print(lo, hi) ... # 1 9
8
9
10 main()
```

Write good functions

Local variables not global

One thing

Name well

Docstrings

Functions: don't alter global vars

- **Anywhere in the program, can change**
 - other function reading one can't guarantee what value it'll get
- **Code is non-local**
 - Rarely sensible to change variables from a functions
- **Makes functions harder to test in isolation**
 - you have to set up the right global state before the function behaves correctly
- **Bugs get non-local**
 - the value causing a failure inside the function was set somewhere else entirely, often far away in the code



Spaghetti Code

Functions: don't alter global vars

• No!

```
1 total = 0 # global
2
3 def add_to_cart(price):
4     global total
5     total += price
6
7 def apply_discount():
8     global total
9     total = total * 0.9
10
11 def print_receipt():
12     print(f"Total: £{total:.2f}")
13
14 def main():
15     # First customer
16     add_to_cart(20)
17     apply_discount()
18     print_receipt()
19
20     # Second customer
21     add_to_cart(5)
22     print_receipt()
23
24 main()
```

• Yes

```
def add_to_cart(total, price):
    return total + price

def apply_discount(total, discount=0.1):
    return total * (1 - discount)

def print_receipt(total):
    print(f"Total: £{total:.2f}")

def main():
    # First customer
    total = 0
    total = add_to_cart(total, 20)
    total = add_to_cart(total, 10)
    total = apply_discount(total)
    print_receipt(total)    # Total: £27.00

    # Second customer
    total = 0
    total = add_to_cart(total, 5)
    print_receipt(total)    # Total: £5.00

main()
```

Functions: do one thing

- **Difficult to name**
 - sign it's doing more than one job
- **Easier to test**
 - one job has one thing to check
 - a function with five jobs needs five different setups to test properly
- **Easier to reuse**
 - can be reused wherever you need to do that thing
 - building blocks, less can go wrong
- **Easier to fix**
 - single-purpose narrows down where to look

No



Yes



Smelly code

- Smells bad!

```
1 def process_scores(raw_scores):
2     scores = [int(s) for s in raw_scores]
3     passed = [s for s in scores if s >= 40]
4     print(f"{len(passed)} out of {len(scores)} passed")
5     with open("results.txt", "a") as f:
6         f.write(f"{passed}\n")
7     return passed
8
9 def main():
10     raw_scores = ["72", "35", "58", "91", "40"]
11     process_scores(raw_scores)
```

- Smells good!

```
1 def parse_scores(raw_scores):
2     return [int(s) for s in raw_scores]
3
4 def passing_scores(scores, threshold=40):
5     return [s for s in scores if s >= threshold]
6
7 def report(passed, total):
8     print(f"{len(passed)} out of {total} passed")
9
10 def save_results(passed):
11     with open("results.txt", "a") as f:
12         f.write(f"{passed}\n")
13
14 def main():
15     raw_scores = ["72", "35", "58", "91", "40"]
16     scores = parse_scores(raw_scores)
17     passed = passing_scores(scores)
18     report(passed, len(scores))
19     save_results(passed)
```

Functions: purposeful names

- **Vague**

```
do_stuff(basket)
```

```
check(text)
```

```
handle(user)
```

```
process2(users)
```

- **Purpose**

```
calculate_total(basket)
```

```
is_valid_email(text)
```

```
send_reminder(user)
```

```
filter_active_users(users)
```

Conclusions

- **Good functions aren't just about correct output — they're about code that's clean, and scalable**
 - A named reusable block that does one job on input and hands back a result
 - Return values, don't print — a function that returns is reusable, one that prints isn't
 - Keep variables local, not global — it makes functions predictable and easy to test
 - Do one thing — easier to name, test, reuse, and fix
 - Name functions for what they do, not vaguely
- **Functions are first taste of modular design *instinct***
 - write once, organise well, reuse
- **Scales up:**
 - **Functions** → **Modules** → **Classes** → **Packages** → **Codebase Architecture**

Thanks!