GCU
Glasgow Caledonian University

From Functions to Class Methods

The big picture: separate spaces

- **Functions, in and out:**
  - arguments in, values back
- **Forgetting:**
  - functions own their own space:
  - each function gets own ‘private’ set of variables
  - same-named variable: used in different places
  - function scope: set fresh every time function runs
  - … ’debris’ cleared away when it finishes
- **Remembering:**
  - a function can hold on to its space instead of letting it be cleared, so it remembers between uses
- **Keeping things together:**
  - putting information and the actions that use it in one place makes an ‘object’ (next lecture)

args **in,** parameters catch values



functions have behaviour

product/result returned **out**

Overview

- **Functions recap**
  - local variables and scope
  - what goes in, what comes out, what stays inside
- **Python can't overload like Java**
  - Java: name re-use (is good)
  - Python: one function instead, flexible impl.
- **Inner functions**
  - helpers tucked inside a function
- **Functions that remember**
  - …and keep things together
- **The idea of objects**
  - actions on data that lasts between uses
- **Classes and class methods (briefly)**







GCU Glasgow Caledonian University

Functions Recap

- Arguments go in, values come back out
- Function variables and ‘scope’

Transactional: arg, param, return!

- **Inputs go in through parameters**
  - `greet()` takes none
  - `add(a, b)` takes two
- **Default on a 'missing' input**
  - `power(5)` uses `exponent=2`
  - `power(5, 3)` overrides it
- **Send the answer back**
  - `return`
    - caller can use that value

      print, store, use generally

```
1   def greet():                    # no inputs
2       return "Hello!"
3
4
5   def add(a, b):                  # two inputs
6       return a + b
7
8
9   def power(base, exponent=2):    # a default
10      return base ** exponent
11
12
13  def main():
14      print(greet())          # Hello!
15      print(add(3, 5))        # 8
16      print(power(5))         # 25  (uses the default)
17      print(power(5, 3))      # 125
18
19
20  if __name__ == "__main__":
21      main()
```

Recap

- **Hands answer back**
  - `add(3, 5)` becomes the value 8
- **Caller decides what to do**
  - store it: `total = add(3, 5)`
  - reuse it: `total * 2` gives 16
  - `print` only shows a value
- **If `add` printed instead of return**
  - total would be `None`
  - `total * 2` would crash
- **Rule of thumb:**
  - functions `return`, `main()` prints

```
1   def add(a, b):
2       return a + b     # hand it back
3
4
5   def main():
6       total = add(3, 5)     # reuse it
7       print(total * 2)      # 16
8
9
10  if __name__ == "__main__":
11      main()
12
```

GCU
Glasgow Caledonian University

Function Overloading

- Same name, different inputs
- Java can – Python can't (quite)

Overloading – Java

- **Same name, different inputs**
  - two methods, both called *area*
- **Java picks the right one**
  - by the number of inputs
  - `area(4.0)` → square
  - `area(5.0, 2.0)` → rectangle
- **One name, many impl.**
  - Good 'name spacing'

```java
public class Shapes {
    static double area(double side) {
        return side * side;
    }

    static double area(double length, double width) {
        return length * width;
    }

    public static void main(String[] args) {
        System.out.println(area(4.0));      // 16.0
        System.out.println(area(5.0, 2.0)); // 10.0
    }
}

```

Overloading – Python?

- **Two functions, same name**
  - one takes a side, the other takes length and width
- **Python: one name, one thing**
  - the second `def` replaces the first
  - only the last `def` survives
- `area(5, 2)`
  - works and gives 10
- `area(4)`
  - crashes, because the one-input version is gone
- **The error tells you which version is left**
  - "missing `'width'`" means Python only knows `area(length, width)`
- **Java would keep both and pick by the inputs**
  - Python can't

```
1   def area(side):
2       return side * side
3
4
5   def area(length, width):  # replaces it!
6       return length * width
7
8
9   def main():
10      print(area(5, 2))   # 10
11      print(area(4))      # crashes on purpose:
12
13
14  if __name__ == "__main__":
15      main()
16
```

Overloading – Python

Glasgow Caledonian University

option 1: a default

```
1   def area(length, width=None):
2       if width is None:      # one value
3           return length * length
4       return length * width  # two values
5
6
7   def main():
8       print(area(4))      # 16  (square)
9       print(area(5, 2))   # 10  (rectangle)
10
11
12  if __name__ == "__main__":
13      main()
14
```

option 2: a branch

```
1   def area(shape, size):
2       if shape == "square":
3           return size * size
4       elif shape == "circle":
5           return 3.14 * size * size
6       else:
7           raise ValueError(shape)
8
9
10  def main():
11      print(area("square", 4))   # 16
12      print(area("circle", 1))   # 3.14
13
14
15  if __name__ == "__main__":
16      main()
17
```

either way: one function, one clear name – `area`

GCU
Glasgow Caledonian University

Scope & Inner Functions

- functions that forget to…
- …functions that remember

Functions that forget

- **Local variables are made fresh…**
  - …every time the function is called
- **Thrown away when it returns**
  - every call starts from scratch
- **Can't be seen from outside**
- **Usually what we want**
  - no surprises between calls

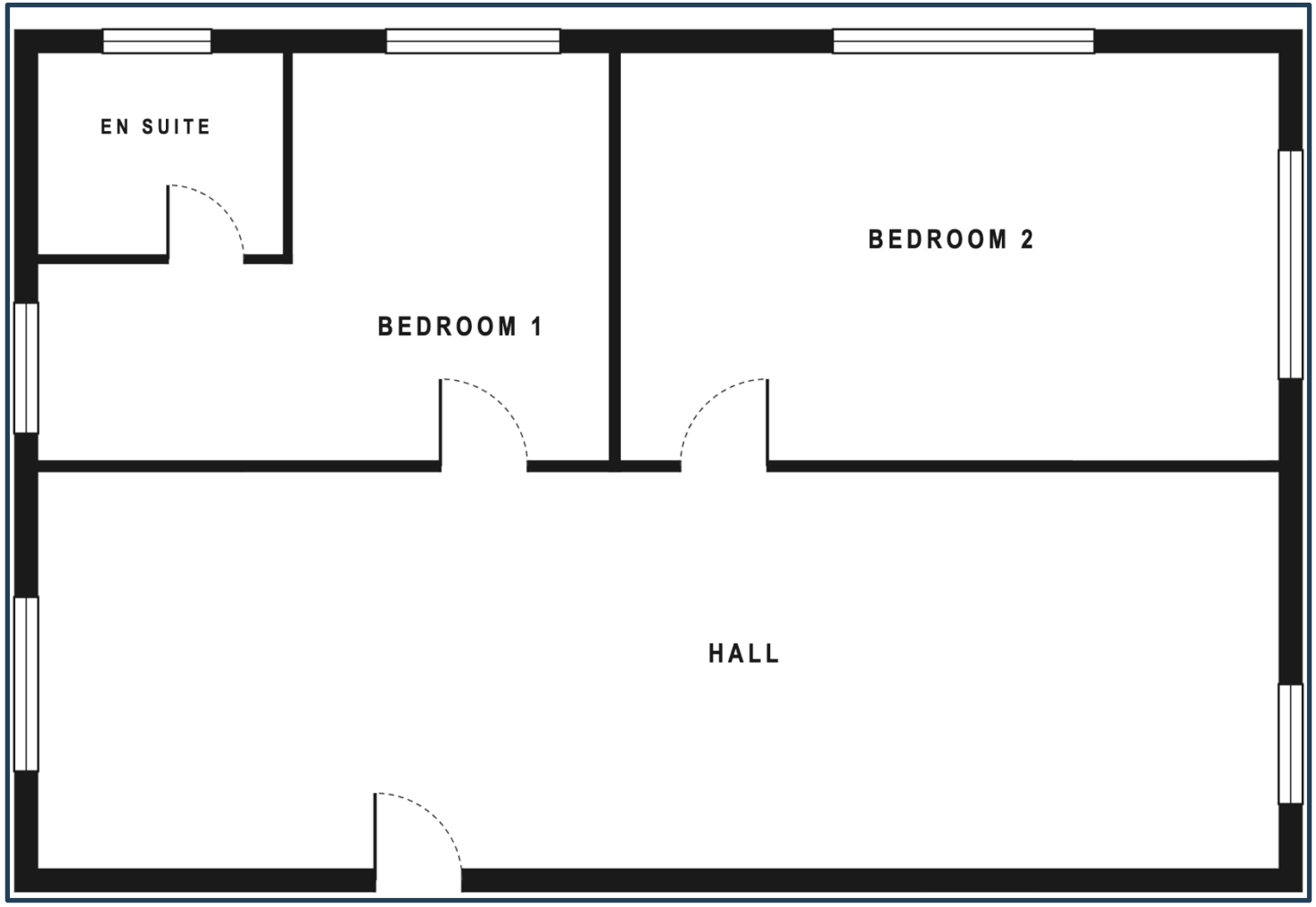
```
1   def count_up():
2       count = 0       # made fresh each call
3       count += 1
4       return count
5
6
7   def main():
8       print(count_up())   # 1
9       print(count_up())   # 1  (forgot!)
10
11      print(count)        # crashes on purpose:
12
13
14  if __name__ == "__main__":
15      main()
16
```

Scope – who can see what

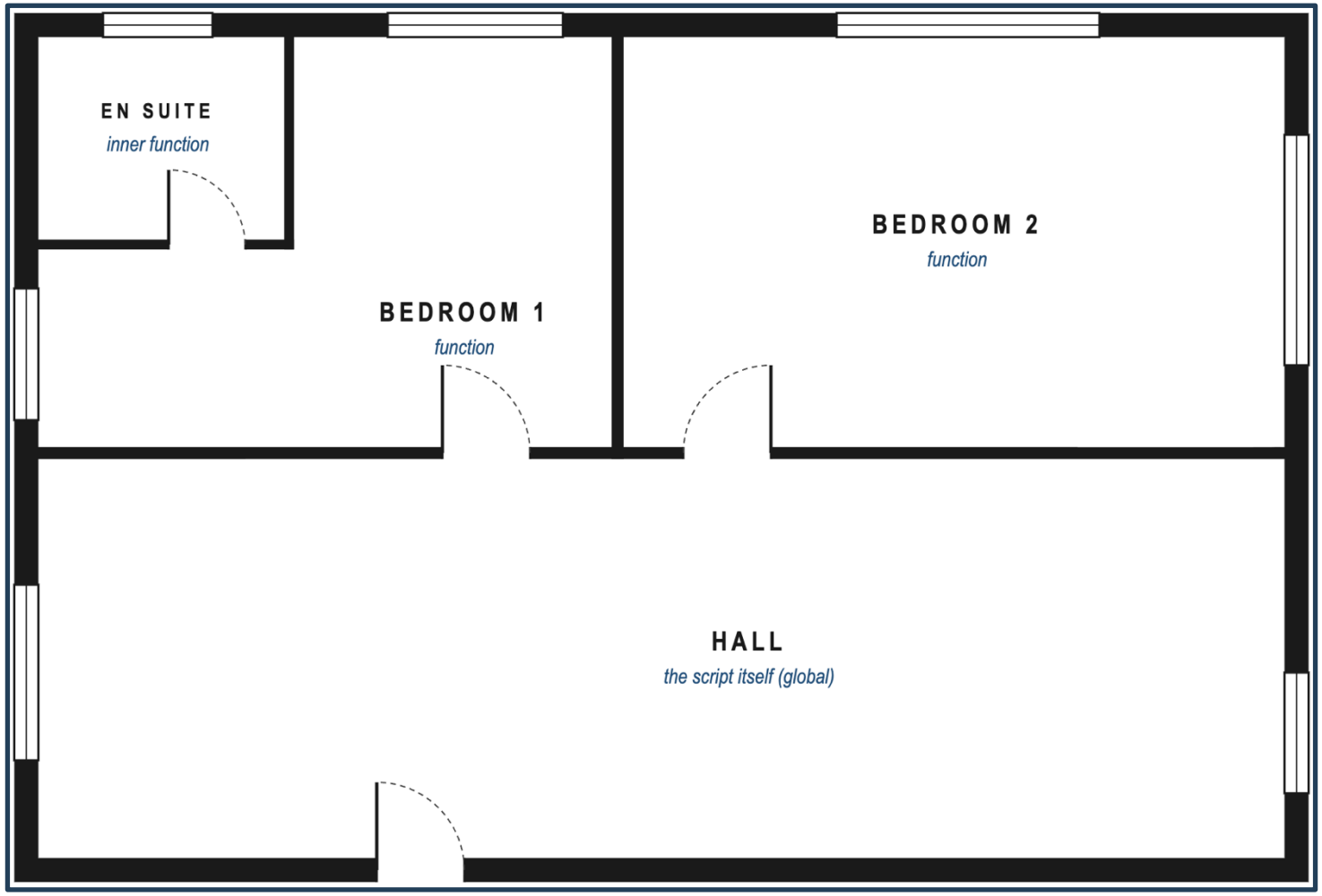
- **Global: made at file level**
  - seen everywhere
- **Local: made inside a function**
  - only seen in there
- **Inner sees outer…**
  - …outer can't see inner
  - trying gives a `NameError`

```
1   def outer():
2       y = "outer"          # local to outer
3
4       def inner():
5           z = "inner"      # local to inner
6           print(x, y, z)   # sees all three
7
8       inner()
9       print(z)             # crashes on purpose:
10
11
12  def main():
13      outer()
14
15
16  if __name__ == "__main__":
17      main()
```

Scope – who can see what



Scope – who can see what



Inner functions

- **Function defined inside another**
  - a small helper
- **Uses the outer variables**
  - no need to pass them in
- **Hidden from outside**
  - `area()` out here → `NameError`
  - hiding helpers = encapsulation

```
1  def describe(length, width):
2
3      def area():                 # inner
4          return length * width
5
6      def perimeter():            # inner
7          return 2 * (length + width)
8
9      a = area()
10     p = perimeter()
11     return f"area {a}, perimeter {p}"
12
13
14 def main():
15     print(describe(5, 2))   # area 10, perimeter 14
16
17     area()                  # crashes on purpose: N
18
19
20 if __name__ == "__main__":
21     main()
22
```

GCU Glasgow Caledonian University

Remembering State

- Functions that forget
- Who can see what
- Functions that remember

Inner functions

- `return` the inner function
  - no brackets ( )
  - hand it back, don't call it
- **Outer variables**
  - kept alive
- `nonlocal`
  - let inner function change the dip
- **Each dunker has memory**
  - data + behaviour together…

```
1   def dairylea_dunker_pack():
2       """The main tub holding the cheese dip"""
3       cheese_dip = "Creamy Dairylea"  #
4
5       def jumbo_tube():
6           """Inner Function: The crunchy stick that reaches out to the outer tub"""
7           # Dipping into the nonlocal cheese!
8           nonlocal cheese_dip
9           print(f"Dunking tube into: {cheese_dip}")
10
11          # Modifying the cheese directly in the outer tub
12          cheese_dip = "Creamy Dairylea (with breadcrumbs left behind)"
13
14      return jumbo_tube
15
16  def main():
17      # Opening a fresh pack
18      dunk = dairylea_dunker_pack()
19
20      # Dip 1
21      dunk()  # Output: Dunking tube into: Creamy Dairylea
22
23      # Dip 2: Accessing modified cheese in the outer scope
24      dunk()  # Output: Dunking tube into: Creamy Dairylea (with breadcrumbs left behind)
25
26
27  if __name__ == "__main__":
28      main()
```



What prints?

```
def make_counter():
    count = 0

    def increment():
        nonlocal count
        count += 1
        return count

    return increment

c = make_counter()
print(c())    # ?
print(c())    # ?
```

Functions that remember

- `return` the inner function
  - no brackets `()`
  - hand it back, don't call it
- **Outer variables**
  - kept alive
- `nonlocal`
  - inner function changes `count`
- **Each counter has memory**
  - data + behaviour together…

```
1   def make_counter():
2       count = 0
3
4       def increment():
5           nonlocal count   # the outer count
6           count += 1
7           return count
8
9       return increment     # no ()
10
11
12  def main():
13      c = make_counter()
14      print(c())   # 1
15      print(c())   # 2  (remembered!)
16
17      d = make_counter()
18      print(d())   # 1  (its own count)
19
20
21  if __name__ == "__main__":
22      main()
```

the proper name for a function that remembers is a closure

Common mistakes

GCU Glasgow Caledonian University

1. forgetting `nonlocal`

```
def make_counter():
    count = 0
    def increment():
        count += 1      # no nonlocal!
        return count
    return increment

c = make_counter()
c()
# UnboundLocalError
```

fix: add `nonlocal count` inside `increment`, so it changes the outer count

2. calling instead of returning

```
def make_counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment()    # () !

c = make_counter()    # c is just 1
c()
# TypeError: 'int' object
#    is not callable
```

fix: `return increment` – no brackets. With brackets only the number gets out, so the counter is forgotten

Similar idea, but as a class

- **Same job as `make_counter`**
- **Data:**
  - `self.count`
- **Behaviour:**
  - `increment()`: class 'method'
- **Each `Counter()` has its own count**
  - next lecture: classes properly

```
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1
        return self.count


def main():
    c = Counter()
    print(c.increment())  # 1
    print(c.increment())  # 2

    d = Counter()         # a second counter
    print(d.increment())  # 1
    print(c.count)        # 2  (c unaffected)

    print("------")
    counter = Counter()
    n = 10
    for _ in range(n):
        print(counter.increment())


if __name__ == "__main__":
    main()

```

Conclusions

- **Python has no overloading**
  - use a default or a branch
- **Local variables are forgotten**
  - when the function returns
- **Inner function**
  - inner sees outer, not the reverse
  - a returned inner function remembers
  - that's a closure: data + behaviour
- **data + behaviour**
  - Classes next week



GCU
Glasgow Caledonian University

Thanks!

questions?