

Loops and lists

Big picture

- **Efficiency**

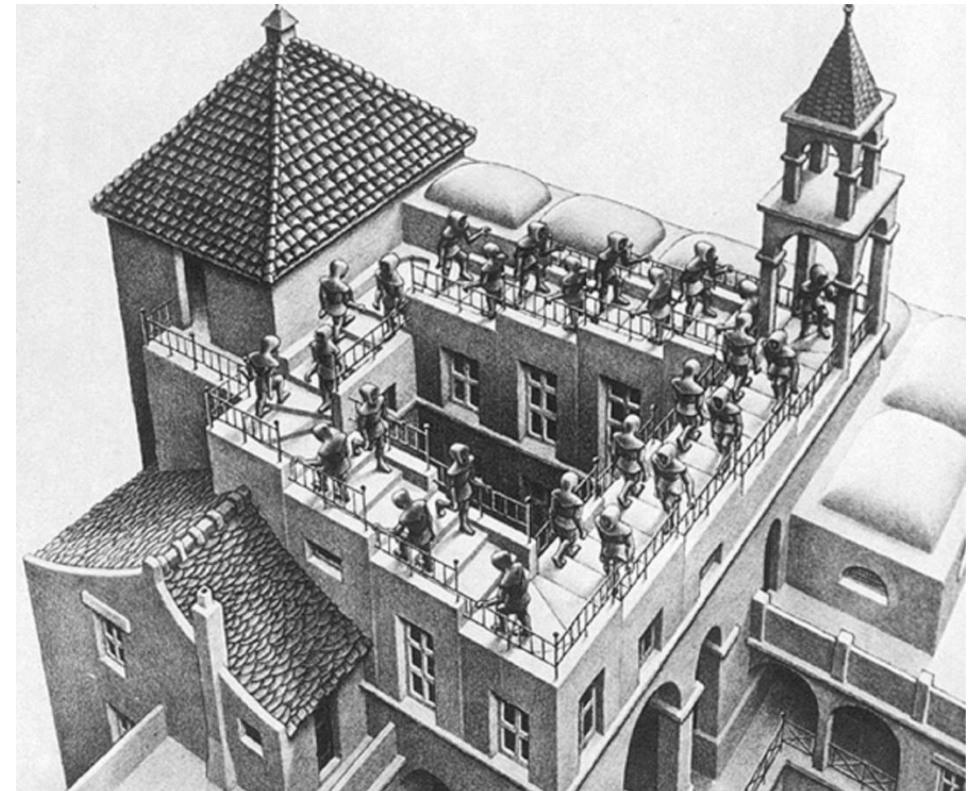
- every programming language needs a way to repeat instructions, because doing the same thing hundreds of times by hand is a programming sin.

- **Problem**

- how do we tell the computer "do this again" without writing it out every time?

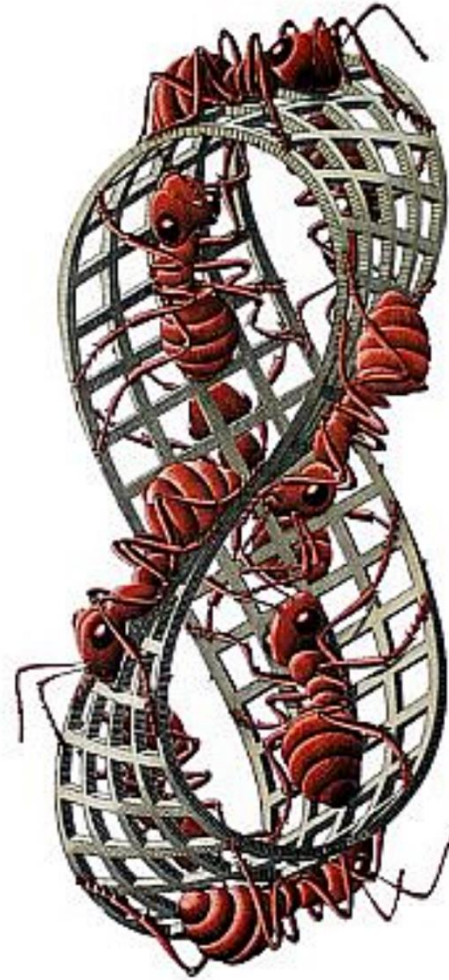
- **Code-repetition**

- an absolute programming sin
- loops are basic mechanism to help us avoid repetition



Overview

- **Motivation**
 - Replace naïve repetition
- **‘while’ loops**
 - Infinity & ‘Control + C’
 - Decrementing, incrementing, conventions
- **‘for’ loops**
 - The `range()` function
 - Underscore convention
- **Lists**
 - Zero-indexing
 - Iterating with `len()`



Loops: motivation

- Naive approach: hard-coding
- Unscalable approach: copy-pasting hard-codes

Lots of coffee orders!

- **The naive approach**

- one code statement per order
- functionally correct but poor, *redundant* code

- **Copy-pasting doesn't scale**

- fine for 3 — not for 50, or 500
- change the message? edit it in every single place

```
print("Order 1 up!")  
print("Order 2 up!")  
print("Order 3 up!")
```

```
print("Order 1 up!")  
print("Order 2 up!")  
print("Order 3 up!")  
# ...paste again for order 4  
# ...again for order 5  
# ...x500 for a busy Saturday?
```

Sum of numbers

- **Adding the numbers manually**
 - works fine, but only because the list happens to have exactly six items
 - bug when the list grows
- **Add a 7th number to the list**
 - total is silently wrong
 - no error is raised — just quietly gives the wrong answer
 - have to remember to go back and add another `total += numbers[6]` line by hand
- **Why this matters**
 - functional correctness depends on keeping two things in sync: the list *and* the manual additions
 - a loop would remove that mental dependency entirely

```
numbers = [4, 8, 15, 16, 23, 42]
total = 0
total += numbers[0]
total += numbers[1]
total += numbers[2]
total += numbers[3]
total += numbers[4]
total += numbers[5]
print(total)
```

```
numbers = [4, 8, 15, 16, 23, 42, 8]
total = 0
total += numbers[0]
total += numbers[1]
total += numbers[2]
total += numbers[3]
total += numbers[4]
total += numbers[5]
# numbers[6] never got added!
print(total)
```

***'for'* loops**

- For loops & lists
- The `range()` function
- The underscore convention

Lots of coffee orders!

- **Loop**

- succinct way to iterate

- **Starting the count at 1**

- `range(n)`

count 0 to $n-1$

- `range(1, n + 1)`

counting begins at 1 and includes n

- **Putting the number in the message**

- an f-string (`f"Order {order_num} up!"`) drops the loop variable straight into the text

- **Same code**

- whether n is 3 or 500 — and now each one is labelled

```
n = 500
for order_num in range(1, n + 1):
    print(f"Order {order_num} up!")
```

Simple 'sum' example

- **The loop itself**

- for every item in the list
- adds each number to total
- no hardcoded indices or missing element

- **Labelling the output**

- `print`

- accepts multiple comma-separated arguments

- space-separated automatically

- **Same for loop code**

- whether the list has 6 numbers or 6,000

```
numbers = [4, 8, 15, 16, 23, 42]
total = 0
for n in numbers:
    total += n
print("Total:", total)
```

$$\sum_{i=1}^n x_i$$

```
x = [4, 8, 15, 16, 23, 42]
total = 0
for i in range(len(x)):
    total += x[i]

print(f"Total: {total}")
```

Specific v General

- **Overly specific, terrible:**

```
print("Order 1 up!")
print("Order 2 up!")
print("Order 3 up!")
# ...paste again for order 4
# ...again for order 5
# ...x500 for a busy Saturday?
```

```
numbers = [4, 8, 15, 16, 23, 42, 8]
total = 0
total += numbers[0]
total += numbers[1]
total += numbers[2]
total += numbers[3]
total += numbers[4]
total += numbers[5]
# numbers[6] never got added!
print(total)
```

- A for loop trades specificity for generality: instead of code that describes six particular numbers, it describes a rule — "do this to each item" — that applies whether there are six or six thousand. The logic no longer depends on how much data there is, only on what should happen to each piece of it.
- The mental shift a loop represents: solving one instance of a problem to solving entire class.

- **Generic, flexible:**

```
n = 500
for order_num in range(1, n + 1):
    print(f"Order {order_num} up!")
```

```
numbers = [4, 8, 15, 16, 23, 42]
total = 0
for n in numbers:
    total += n
print("Total:", total)
```

Pythonic underscore

- **The loop variable**

- a for loop needs a variable to count with

- **The underscore**

- convention when the value is never used in the loop
- signal: this variable is unused

- **This is not necessary**

-but is a convention!

```
for _ in range(3):  
    print("Next customer, please!")
```

***‘while’* loops**

- The while paradigm
- Infinite loops & Control-C
- Decrementing, incrementing, conventions
- While vs for
- User input validation example

Lots of coffee orders!

- The **while** keyword

- repeatedly ask question
 - ‘is the order number less than or equal to zero’?
- to decide whether to continue
- answer $\in \{True, False\}$
- `order_num`
 - starts at 1;
 - the loop keeps going while `order_num` less than or equal (`<=`) to `n`

```
n = 500
order_num = 1
while order_num <= n:
    print(f"Order {order_num} up!")
```

Infinite loops

- **The bug**

- `order_num` never changes — the loop runs forever

- **The fix (for now)**

- Control-C in the terminal cancels a runaway program
- Returns control to Terminal

```
n = 500
order_num = 1
while order_num <= n:
    print(f"Order {order_num} up!")
```

Decrementing, incrementing, and conventions

- **Decrementing**

- `order_num -= 1` each iteration, until it hits zero

- **Incrementing**

- or start at 1 and count upward to n instead

```
order_num = n
while order_num > 0:
    print(f"Order {order_num} up!")
    order_num -= 1
```

```
# or count up instead
order_num = 1
while order_num <= n:
    print(f"Order {order_num} up!")
    order_num += 1
```

'while' vs 'for'

- **Same coffee orders, two ways**

- `for` handles the counting automatically
- `while` needs it done by hand: initialize, check, update

- **Why bother with *while*?**

- `for` loops over a known, finite range
- `while` loops until a condition changes — even when you don't know how many times in advance

for loop

```
n = 500
for order_num in range(1, n + 1):
    print(f"Order {order_num} up!")
```

while loop

```
n = 500
order_num = 1
while order_num <= n:
    print(f"Order {order_num} up!")
    order_num += 1
```

User-input validation

- **Guarantee valid input**

- `while True:` creates infinite loop

- **‘continue’ vs ‘break’**

- `continue` to keep looping

- `break` infinity to exit once valid

- **Logic**

- the order number must be a digit
 - program logic captures problem domain constraints

```
while True:
    order_num = input("Order number: ")
    if not order_num.isdigit():
        continue
    order_num = int(order_num)
    if order_num > 0:
        break
```

Lists

- Zero-indexing
- Iterating with len()

Zero-indexing

- **Lists can hold richer data**
 - e.g. a list of strings naming tonight's setlist
- **Zero-indexing**
 - the first element sits at index 0

```
setlist = ["Intro", "Wildfire", "Encore"]  
print(setlist[0])  # Intro  
print(setlist[2])  # Encore
```

$$\sum_{i=1}^n x_i$$

```
x = [4, 8, 15, 16, 23, 42]  
total = 0  
for i in range(len(x)):  
    total += x[i]  
  
print(f"Total: {total}")
```

$$\sum_{i=0}^{n-1} x_i$$

Iterating lists & using 'len()'

- **Step through item-by-item**

- `for song in setlist`

- **Or generate indices dynamically**

- `range(len(setlist))` sizes itself to the list

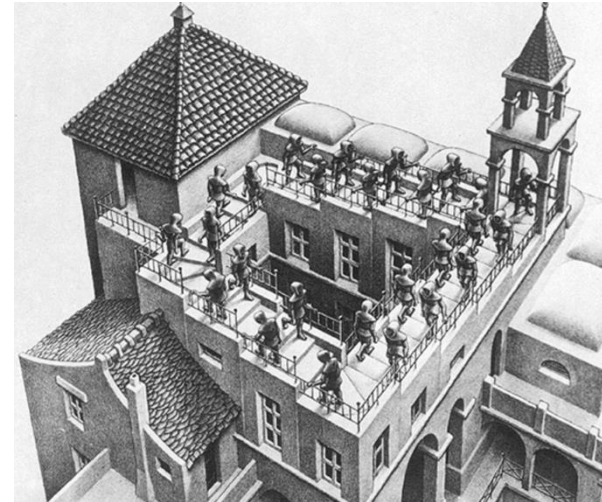
- **Depends on what you want to achieve**

```
for song in setlist:  
    print(song)
```

```
for i in range(len(setlist)):  
    print(i, setlist[i])
```

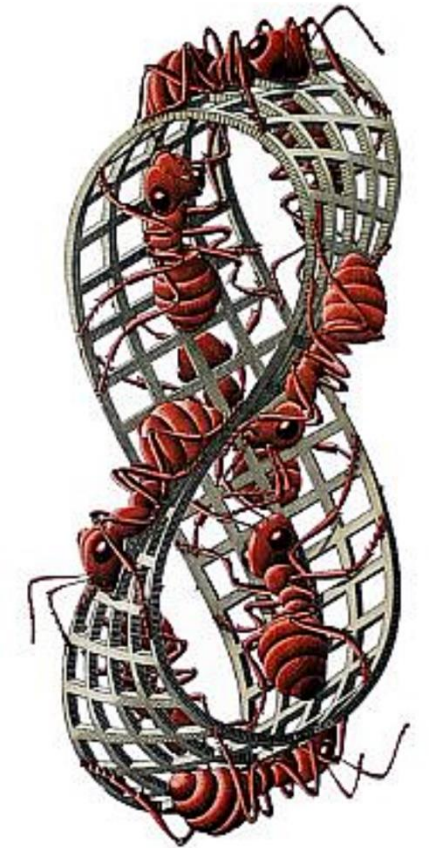
Conclusions

- **Where we started**
 - repeating instructions by hand doesn't scale.
 - copy-pasting / manual list addition brake as data grows
- **Loops buy you**
 - code that describes a rule ("do this to each item")
 - flexible, more generic code
- **The 'while' loop**
 - the condition is true, while code executes
 - stopping condition not necessarily known advance
- **The 'for' loop**
 - use when you're working through a known range or collection
- **Lists (in relation to 'for')**
 - zero-indexing, `len()`, `range()` with `for` loops
- **Solving specific instances → solving more general cases**
 - solving one instance of a problem → solving the entire class of it



```
x = [4, 8, 15, 16, 23, 42]
total = 0
for i in range(len(x)):
    total += x[i]

print(f"Total: {total}")
```



Thanks!