**Imperative = “do this step, then that step” vs. FP = “describe what I want”**

|  |  |  |
| --- | --- | --- |
| **Example** | **Imperative** | **Functional** |
| List Transformation (Squaring Numbers) | numbers = [1, 2, 3, 4]  squared\_numbers = []  for num in numbers:  squared\_numbers.append(num \* num) | numbers = [1, 2, 3, 4]  squared\_numbers = list(map(lambda x: x \* x, numbers)) |
| Sum of squares of a list | nums = [1, 2, 3, 4, 5] squares = [] for n in nums:  squares.append(n \* n) result = sum(squares) | from functools import reduce nums = [1, 2, 3, 4, 5] result = reduce(lambda x, y: x + y, map(lambda n: n \* n, nums)) |
| Combining Filtering and Mapping | numbers = [1, 2, 3, 4, 5, 6]  temp\_list = []  for number in numbers:  if number % 2 == 0:  temp\_list.append(number)  squared\_even\_numbers = []  for number in temp\_list:  squared\_even\_numbers.append(number \*\* 2) | numbers = [1, 2, 3, 4, 5, 6]  squared\_even\_numbers = [n \*\* 2 for n in numbers if n % 2 == 0] |
| Combining filter/map/reduce | def sum\_of\_even\_squares\_imperative(numbers):  result = 0  for num in numbers:  if num % 2 == 0:  result += num \*\* 2  return result | from functools import reduce  def sum\_of\_squares\_of\_even\_numbers(numbers):  even\_numbers = filter(lambda x: x % 2 == 0, numbers)  squares = map(lambda x: x \*\* 2, even\_numbers)  sum\_of\_squares = reduce(lambda x, y: x + y, squares, 0)  return sum\_of\_squares |
| Filtering even numbers | nums = [1, 2, 3, 4, 5, 6] evens = [] for n in nums:  if n % 2 == 0:  evens.append(n) | nums = [1, 2, 3, 4, 5, 6] evens = list(filter(lambda n: n % 2 == 0, nums)) |
| Convert strings to uppercase | words = ["hello", "world"] upper\_words = [] for w in words:  upper\_words.append(w.upper()) | words = ["hello", "world"] upper\_words = list(map(str.upper, words)) |
| Word frequency count | text = "apple banana apple orange banana apple" counts = {} for word in text.split():  if word not in counts:  counts[word] = 0  counts[word] += 1 print(counts) # {'apple': 3, 'banana': 2, 'orange': 1} | from collections import Counter text = "apple banana apple orange banana apple" counts = Counter(text.split()) |
| Fibonacci seq | n = 7 fib = [0, 1] for i in range(2, n):  fib.append(fib[-1] + fib[-2]) | from functools import reduce n = 7 fib = reduce(lambda acc, \_: acc + [acc[-1] + acc[-2]], range(n-2), [0, 1]) |
| Factorial | def factorial\_iter(n):  result = 1  for i in range(1, n + 1):  result \*= i  return result | def factorial\_rec(n):  return 1 if n == 0 else n \* factorial\_rec(n - 1) |
| Sum nested lists | nested = [[1, 2], [3, 4], [5, 6]] total = 0 for sub in nested:  for num in sub:  total += num | nested = [[1, 2], [3, 4], [5, 6]] total = sum(num for sub in nested for num in sub) |
| Grouping by key | items = [('fruit', 'apple'), ('fruit', 'banana'), ('veg', 'carrot')] grouped = {} for cat, item in items:  grouped.setdefault(cat, []).append(item) | from itertools import groupby items = sorted(items, key=lambda x: x[0]) grouped = {k: [item for \_, item in grp] for k, grp in groupby(items, key=lambda x: x[0])} |
| Sliding window average | values = [1, 2, 3, 4, 5] window = 3 averages = [] for i in range(len(values) - window + 1):  window\_slice = values[i:i + window]  averages.append(sum(window\_slice) / window) | values = [1, 2, 3, 4, 5] window = 3 averages = [(sum(values[i:i + window]) / window) for i in range(len(values) - window + 1)] |
| Chaining string transform | sentence = "hello world" parts = sentence.split() parts = [p.capitalize() for p in parts] result = "-".join(parts) | result = "-".join(word.capitalize() for word in "hello world".split()) |
| Flatten nested list (1 level) | deep = [1, [2, 3], [4, [5, 6]], 7] flat = [] for item in deep:  if isinstance(item, list):  flat.extend(item)  else:  flat.append(item) | flat = [x for item in deep for x in (item if isinstance(item, list) else [item])] |
| Filtering with multi conditions | nums = list(range(20)) res = [] for n in nums:  if n % 2 == 0 and n > 10:  res.append(n) | res = list(filter(lambda n: n % 2 == 0 and n > 10, range(20))) |
| Limit results | nums = range(100) res = [] count = 0 for n in nums:  if n % 5 == 0:  res.append(n)  count += 1  if count == 3:  break | from itertools import islice res = list(islice(filter(lambda n: n % 5 == 0, range(100)), 3)) |
| Running sums | nums = [1, 2, 3, 4] running = [] total = 0 for n in nums:  total += n  running.append(total) | from itertools import accumulate nums = [1, 2, 3, 4] running = list(accumulate(nums)) |
| Error-safe parsing | values = ["1", "two", "3", "four"] parsed = [] for v in values:  try:  parsed.append(int(v))  except ValueError:  parsed.append(None) | def safe\_int(s):  try:  return int(s)  except ValueError:  return None parsed = list(map(safe\_int, values)) |
| Combine lists into dict | keys = ['a', 'b', 'c'] values = [1, 2, 3] d = {} for i in range(len(keys)):  d[keys[i]] = values[i] | d = dict(zip(keys, values)) |
| Partitioning list | nums = range(10) evens, odds = [], [] for n in nums:  (evens if n % 2 == 0 else odds).append(n) | evens = list(filter(lambda x: x % 2 == 0, range(10))) odds = list(filter(lambda x: x % 2 != 0, range(10))) |
| Unique + sorted | nums = [3, 1, 2, 3, 2] unique\_sorted = sorted(set(nums)) | unique\_sorted = sorted({\*nums}) |
| Memoization recursion | memo = {} def fib(n):  if n in memo:  return memo[n]  if n < 2:  memo[n] = n  else:  memo[n] = fib(n-1) + fib(n-2)  return memo[n] | from functools import lru\_cache @lru\_cache(maxsize=None) def fib(n):  return n if n < 2 else fib(n-1) + fib(n-2) |