


```

152 #
153 #
154 #
155 #
156 #
157 #
158 #
159
160 def any(a: list) -> Any:
161     "Return a random item."
162     return a[anywhere(a)]
163
164 def anywhere(a: list) -> int:
165     "Return a random index of list 'a'."
166     return random.randint(0, len(a)-1)
167
168 big = sys.maxsize
169
170 def atom(x):
171     "Return a number or trimmed string."
172     x = x.strip()
173     if x == "True":
174         return True
175     elif x == "False":
176         return False
177     else:
178         try:
179             return int(x)
180         except:
181             try:
182                 return float(x)
183             except:
184                 return x.strip()
185
186 def demo(do, all):
187     "Maybe run a demo, if we want it, resetting random seed first."
188     todo = dir(all)
189     if do and do != "all":
190         todo = [x for x in dir(all) if x.startswith(do)]
191     for one in todo:
192         fun = all.__dict__.get(one, "")
193         if type(fun) == type(demo):
194             random.seed(the.seed)
195             doc = re.sub(r'\n+', "\n", fun.__doc__ or "")
196             try:
197                 fun()
198                 print("PASS.", doc)
199             except Exception as e:
200                 all.fail += 1
201                 if the.cautious:
202                     traceback.print_exc()
203                     exit(1)
204             else:
205                 print("FAIL:", doc, e)
206         exit(all.fail)
207
208 def file(f):
209     "Iterator. Returns one row at a time, as cells."
210     with open(f) as fp:
211         for line in fp:
212             line = re.sub(r'([\\n\\r\\`\\`])#.*', '', line)
213             if line:
214                 yield [atom(cell.strip()) for cell in line.split(",")]
215
216 def first(a: list) -> Any:
217     "Return first item."
218     return a[0]
219
220
221 def merge(b4: list) -> list:
222     "While we can find similar adjacent things, merge them."
223     j, n, now = -1, len(b4), []
224     while j < n-1:
225         j += 1
226         a = b4[j]
227         if j < n-2:
228             if merged := a.merge(b4[j+1]):
229                 a = merged
230                 j += 1 # we will continue, after missing one
231             now += [a]
232         # if 'now' is same size as 'b4', look for any other merges.
233     return b4 if len(now) == len(b4) else merge(now)
234
235 class o(object):
236     "Class that can pretty print its slots, with fast inits."
237     def __init__(i, **d): i.__dict__.update(**d)
238
239     def __repr__(i):
240         pre = i.__class__.__name__ if isinstance(i, o) else ""
241         return pre+str(
242             {k: v for k, v in sorted(i.__dict__.items()) if str(k)[0] != "_"})
243
244 def options(doc: str) -> o:
245     """Convert 'doc' to options dictionary using command line args.
246     Args cause two 'shortands': (1) boolean flags have no arguments (and mentioning
247     those on the command line means 'flip the default value'; (2) args need only
248     mention the first few of a key (e.g. -s is enough to select for -seed)."""
249     d = {}
250     for line in doc.splitlines():
251         if line and line.startswith(" -"):
252             # get 1st, last word on each line
253             key, _ = line.strip()[1:].split(" ")
254             for j, flag in enumerate(sys.argv):
255                 if flag and flag[0] == "-" and key.startswith(flag[1:]):
256                     x = "True" if x == "False" else (
257                         "False" if x == "True" else sys.argv[j+1])
258             d[key] = atom(x)
259         if d["help"]:
260             exit(print(re.sub(r'\n#.*', "", doc, flags=re.S)))
261     return o(**d)
262
263 def r() -> float:
264     "Return random number 0..1"
265     return random.random()
266
267 def rn(x: float, n=3) -> float:
268     "Round a number to three decimals."
269     return round(x, n)
270
271 def rn(a: list, n=3) -> list:
272     "Round a list of numbers to three decimals."
273     return [rn(x, n=n) for x in a]
274
275 def second(a: list) -> Any:
276     "Return second item."
277     return a[1]

```

```

278 #
279 #
280 #
281 #
282 #
283 #
284 #
285
286 #
287 #
288 #
289 #
290 #
291
292 class Span(o):
293     """Given two 'Sample's and some 'x' range 'lo..hi'.
294     a 'Span' holds often that range appears in each 'Sample'."""
295     def __init__(i, col, lo, hi, ys=None):
296         i.col, i.lo, i.hi, i.ys = col, lo, hi, ys or Sym()
297
298     def add(i, x: float, y: Any, inc=1) -> None:
299         "y is a label identifying one 'Sample' or another."
300         i.lo = min(x, i.lo)
301         i.hi = max(x, i.hi)
302         i.ys.add(y, inc)
303
304     def merge(i, j): # -> Span|None
305         "If the merged span is simpler, return that merge."
306         a, b, c = i.ys, j.ys, i.ys.merge(j.ys)
307         if (i.ys.n == 0 or j.ys.n == 0 or
308             c.div()*.99 <= (a.n*a.div() + b.n*b.div())/(a.n + b.n)):
309             return Span(i.col, min(i.lo, j.lo), max(i.hi, j.hi), ys=c)
310
311     def selects(i, row: list) -> bool:
312         "True if the range accepts the row."
313         x = row[i.col.at]
314         return x == "?" or i.lo <= x and x < i.hi
315
316     def show(i, positive=True) -> None:
317         "Show the range."
318         txt = i.col.txt
319         if positive:
320             if i.lo == i.hi:
321                 return f"[{txt}] == {i.lo}"
322             elif i.lo == -big:
323                 return f"[{txt}] < {i.hi}"
324             elif i.hi == big:
325                 return f"[{txt}] >= {i.lo}"
326             else:
327                 return f"[{i.lo}] <= [{txt}] < {i.hi}"
328         else:
329             if i.lo == i.hi:
330                 return f"[{txt}] != {i.lo}"
331             elif i.lo == -big:
332                 return f"[{txt}] >= {i.hi}"
333             elif i.hi == big:
334                 return f"[{txt}] < {i.lo}"
335             else:
336                 return f"[{txt}] < {i.lo} or {txt] >= {i.hi}"
337
338     def support(i) -> float:
339         "Returns 0..1."
340         return i.ys.n / i.col.n
341
342 @staticmethod
343 def sort(spans: list) -> list:
344     "Good spans have large support and low diversity."
345     divs, supports = Num(), Num()
346     def sn(s): return supports.norm(s.support())
347     def dn(s): return divs.norm(s.ys.div())
348     def f(s): return ((1 - sn(s))**2 + dn(s)**2)**.5/2**5
349     for s in spans:
350         divs.add(s.ys.div())
351         supports.add(s.support())
352     return sorted(spans, key=f)
353
354 #
355 #
356 #
357
358 class Col(o):
359     "Summarize columns."
360     def __init__(i, at=0, txt=""):
361         i.n, i.at, i.txt, i.w = 0, at, txt, (-1 if "-" in txt else 1)
362
363     def dist(i, x: Any, y: Any) -> float:
364         return 1 if x == "?" and y == "?" else i.dist1(x, y)
365
366 #
367 #
368 #
369 #
370
371 class Sym(Col):
372     "Summarize symbolic columns."
373     def __init__(i, **kw):
374         super().__init__(**kw)
375         i.has, i.mode, i.most = {}, None, 0
376
377     def add(i, x: str, inc: int = 1) -> str:
378         "Update symbol counts in 'has', updating 'mode' as we go."
379         if x != "?":
380             i.n += inc
381             tmp = i.has[x] = inc + i.has.get(x, 0)
382             if tmp > i.most:
383                 i.most, i.mode = tmp, x
384         return x
385
386     def dist(i, x: str, y: str) -> float:
387         "Distance between two symbols."
388         return 0 if x == y else 1
389
390     def div(i):
391         "Return diversity of this distribution (using entropy)."
392         def p(x): return x / (1E-31 + i.n)
393         return sum((-p(x)*math.log(p(x), 2) for x in i.has.values() )
394
395     def merge(i, j):
396         "Merge two 'Sym's."
397         k = Sym(at=i.at, txt=i.txt)
398         for x, n in i.has.items():
399             k.add(x, n)
400         for x, n in j.has.items():
401             k.add(x, n)
402         return k
403
404     def mid(i):
405         "Return central tendency of this distribution (using mode)."
406         return i.mode
407
408     def spans(i, j, out):
409         """For each symbol in 'i' and 'j', count the
410         number of times we see it on either side."""
411         xys = [(x, "this", n) for x, n in i.has.items()] + [
412             (x, "that", n) for x, n in j.has.items()]
413         one, last = None, None

```

```

414 all = []
415 for x, y, n in sorted(xys, key=first):
416     if x != last:
417         last = x
418         one = Span(i, x, x)
419         all += [one]
420     one.add(x, y, n)
421     if len(all) > 1:
422         out += all
423 #
424 #
425 #
426 #
427 class Num(Col):
428     "Summarize numeric columns."
429     def __init__(i, **kw):
430         super().__init__(**kw)
431         i._all, i.lo, i.hi, i.max, i.ok = [], 1E32, -1E32, the.Max, False
432
433     def add(i, x: float, inc=1):
434         "Reservoir sampler. If '_all' is full, sometimes replace an item at random."
435         if x != "":
436             i.n += inc
437             i.lo = min(x, i.lo)
438             i.hi = max(x, i.hi)
439             if len(i._all) < i.max:
440                 i.ok = False
441                 i._all += [x]
442             elif r() < i.max/i.n:
443                 i.ok = False
444                 i._all[anywhere(i._all)] = x
445         return x
446
447     def all(i):
448         "Return '_all', sorted."
449         if not i.ok:
450             i.ok = True
451             i._all.sort()
452         return i._all
453
454     def dist1(i, x, y):
455         if x == "":
456             y = i.norm(y)
457             x = (1 if y < .5 else 0)
458         elif y == "":
459             x = i.norm(x)
460             y = (1 if x < .5 else 0)
461         else:
462             x, y = i.norm(x), i.norm(y)
463         return abs(x-y)
464
465     def div(i):
466         """Report the diversity of this distribution (using standard deviation).
467         &pm;2, 2.56, 3 &sigma;: is 66.90/95%, of the mass. 2&sigma;: So one
468         standard deviation is (90-10)th divide by 2.4 times &sigma;."""
469         return (i.per(.9) - i.per(.1)) / 2.56
470
471     def merge(i, j):
472         "Return two 'Num's."
473         k = Num(at=i.at, txt=i.txt)
474         for x in i._all:
475             k.add(x)
476         for x in j._all:
477             k.add(x)
478         return k
479
480     def mid(i):
481         "Return central tendency of this distribution (using median)."
482         return i.per(.5)
483
484     def norm(i, x):
485         "Normalize 'x' to the range 0..1."
486         return 0 if i.hi-i.lo < 1E-9 else (x-i.lo)/(i.hi-i.lo)
487
488     def per(i, p: float = .5) -> float:
489         "Return the p-th ranked item."
490         a = i.all()
491         return a[ int(p*len(a)) ]
492
493     def spans(i, j, out):
494         """Divide the whole space 'lo' to 'hi' into, say, 'xsmall'=16 bin,
495         then count the number of times we the bin on other side.
496         Then merge similar adjacent bins."""
497         lo = min(i.lo, j.lo)
498         hi = max(i.hi, j.hi)
499         gap = (hi-lo) / (6/the.xsmall)
500         xys = [(x, "this", 1) for x in i._all] + [
501             (x, "that", 1) for x in j._all]
502         one = Span(i, lo, lo)
503         all = [one]
504         for x, y, n in sorted(xys, key=first):
505             if one.hi - one.lo > gap:
506                 one = Span(i, one.hi, x)
507                 all += [one]
508             one.add(x, y, n)
509         all = merge(all)
510         all[0].lo = -big
511         all[-1].hi = big
512         if len(all) > 1:
513             out += all
514 #
515 #
516 #
517 #
518 #
519 #
520 class Explain(o):
521     "Tree with 'yes','no' branches for samples that do/do not match a 'span'."
522     def __init__(i, here):
523         i.here, i.span, i.yes, i.no = here, None, None, None
524
525     def show(i, pre=""):
526         if not pre:
527             tmp = i.here.mid(i.here.y)
528             print(f"{'':40}: {len(i.here.rows):5}: {tmp}")
529         if i.yes:
530             s = f"{pre}{i.span.show(True)}"
531             tmp = i.yes.here.mid(i.yes.here.y)
532             print(f"{'s':40}: {len(i.yes.here.rows):5}: {tmp}")
533             i.yes.show(pre + "|.")
534         if i.no:
535             s = f"{pre}{i.span.show(False)}"
536             tmp = i.no.here.mid(i.no.here.y)
537             print(f"{'s':40}: {len(i.no.here.rows):5}: {tmp}")
538             i.no.show(pre + "|.")
539 #
540 #
541 #
542 #
543 #
544 #
545 class Cluster(o):
546     "Tree with 'left','right' samples, broken at median between far points."
547     def __init__(i, here, x=None, y=None, c=None, mid=None):
548         i.here, i.x, i.y, i.c, i.mid, i.left, i.right = here, x, y, c, mid, None, No
549 ne

```

```

549
550 def show(i, pre=""):
551     s = f"{pre:40}: {len(i.here.rows):5}"
552     print(f"{'s':40}: {len(i.here.rows):5}: {i.here.mid(i.here.y)}")
553     for kid in [i.left, i.right]:
554         if kid:
555             kid.show(pre + "|.")
556 #
557 #
558 #
559 #
560 #
561 #
562 class Sample(o):
563     "Load, then manage, a set of examples."
564     def __init__(i, inits=[]):
565         i.rows, i.cols, i.x, i.y, i.klass = [], [], [], [], None
566         if str == type(inits):
567             [i.add(row) for row in file(inits)]
568         if list == type(inits):
569             [i.add(row) for row in inits]
570
571     def add(i, a):
572         def col(at, txt):
573             what = Num if txt[0].isupper() else Sym
574             now = what(at=at, txt=txt)
575             where = i.y if "+" in txt or "-" in txt or "!" in txt else i.x
576             if txt[-1] != " ":
577                 where += [now]
578             if "*" in txt:
579                 i.klass = now
580             return now
581         # -----
582         if i.cols:
583             i.rows += [[col.add(a[col.at]) for col in i.cols]]
584         else:
585             i.cols = [col(at, txt) for at, txt in enumerate(a)]
586
587     def clone(i, inits=[]):
588         out = Sample()
589         out.add([col.txt for col in i.cols])
590         [out.add(x) for x in inits]
591         return out
592
593     def cluster(i, top=None):
594         """Split the data using random projections. Find the span that most
595         separates the data. Divide data on that span."""
596         here = Cluster(i)
597         top = top or i
598         if len(i.rows) >= 2*(len(top.rows)**the.enough):
599             left, right, x, y, c, mid = i.half(top)
600             if len(left.rows) < len(i.rows):
601                 here = Cluster(i, x, y, c, mid)
602             here.left = left.cluster(top)
603             here.right = right.cluster(top)
604             return here
605
606     def dist(i, x, y):
607         d = sum( col.dist(x[col.at], y[col.at])**the.p for col in i.x )
608         return (d/len(i.x)) ** (1/the.p)
609
610     def div(i, cols=None):
611         return [col.div() for col in (cols or i.all)]
612
613     def far(i, x, rows=None):
614         tmp = sorted([(i.dist(x, y), y) for y in (rows or i.rows)], key=first)
615         return tmp[ int(len(tmp)*the.far) ]
616
617     def half(i, top=None):
618         "Using two faraway points 'x,y' break data at median distance."
619         some = i.rows if len(
620             i.rows) < the.Some else random.choices(i.rows, k=the.Some)
621         top = top or i
622         w = any(some)
623         _, x = top.far(w, some)
624         c, y = top.far(x, some)
625         tmp = [r for _, r in sorted([(top.proj(r, x, y, c), r)
626                                     for r in i.rows], key=first))]
627         mid = len(tmp)//2
628         return i.clone(tmp[:mid]), i.clone(tmp[mid:]), x, y, c, tmp[mid]
629
630     def mid(i, cols=None):
631         return [col.mid() for col in (cols or i.all)]
632
633     def proj(i, row, x, y, c):
634         "Find the distance of a 'row' on a line between 'x' and 'y'."
635         a = i.dist(row, x)
636         b = i.dist(row, y)
637         return (a**2 + c**2 - b**2) / (2*c)
638
639     def xplain(i, top=None):
640         """Split the data using random projections. Find the span that most
641         separates the data. Divide data on that span."""
642         here = Explain(i)
643         top = top or i
644         tiny = len(top.rows)**the.enough
645         if len(i.rows) >= 2*tiny:
646             left, right, *_ = i.half(top)
647             spans = []
648             [lcol.spans(rcol, spans) for lcol, rcol in zip(left.x, right.x)]
649             if len(spans) > 0:
650                 here.span = Span.sort(spans)[0]
651                 yes, no = i.clone(), i.clone()
652                 [(yes if here.span.selects(row) else no).add(row) for row in i.rows]
653                 if tiny <= len(yes.rows) < len(i.rows):
654                     here.yes = yes.xplain(top=top)
655                 if tiny <= len(no.rows) < len(i.rows):
656                     here.no = no.xplain(top=top)
657             return here

```

```

658 #
659 #
660 #
661 #
662 #
663 #
664 #
665
666 class Demos:
667     "Possible start-up actions."
668     fails = 0
669
670     def opt():
671         "show the config"
672         [print(f"{k}>10}={v}") for k, v in the.__dict__.items()]
673
674     def seed():
675         "seed"
676         assert .494 <= r() <= .495
677
678     def num():
679         "check 'Num'."
680         n = Num()
681         for _ in range(100):
682             n.add(r())
683         assert .30 <= n.div() <= .31, "in range"
684
685     def sym():
686         "check 'Sym'."
687         s = Sym()
688         for x in "aaaabbc":
689             s.add(x)
690         assert 1.37 <= s.div() <= 1.38, "entropy"
691         assert 'a' == s.mid(), "mode"
692
693     def rows():
694         "count rows in a file."
695         assert 399 == len([row for row in file(the.data)])
696
697     def sample():
698         "sampling"
699         s = Sample(the.data)
700         assert 398 == len(s.rows), "length of rows"
701         assert 249 == s.x[-1].has[1], "symbol counts"
702
703     def dist():
704         "distance between rows"
705         s = Sample(the.data)
706         assert .84 <= s.dist(s.rows[1], s.rows[-1]) <= .842
707
708     def far():
709         "distant items"
710         s = Sample(the.data)
711         for _ in range(32):
712             a, _ = s.far(any(s.rows))
713             assert a > .5, "large?"
714
715     def clone():
716         "cloning"
717         s = Sample(the.data)
718         s1 = s.clone(s.rows)
719         d1, d2 = s.x[0].__dict__, s1.x[0].__dict__
720         for k, v in d1.items():
721             assert d2[k] == v, "clone test"
722
723     def half():
724         "divide data in two"
725         s = Sample(the.data)
726         s1, s2, *_ = s.half()
727         print(s1.mid(s1.y))
728         print(s2.mid(s2.y))
729
730     def cluster():
731         "divide data in two"
732         s = Sample(the.data)
733         s.cluster().show()
734         print("")
735
736     def xplain():
737         "divide data in two"
738         s = Sample(the.data)
739         s.xplain().show()
740         print("")
741
742 # -----
743 the = options(__doc__)
744 if __name__ == "__main__":
745     demo(the.todo, Demos)
746
747 """
748 all config local to Sample
749 Example class
750 """

```