

Package **parcel**

This package maps the boundaries of properties from survey descriptions.

It starts from deeds as pdfs, and parses them for numerical information.

The package is especially desgined to deal with 19th-century and other old deeds, since old property descriptions are often repeated in recent deeds, and historical boundaries can be important for property issues.

The package includes code for the following:

Optical Character Recognition

- Since official deeds are in image format, often made from scans of copies, the package transcribes them using tesseract (by default) or easyocr.
- The initial transcriptions may be low-quality because of low-quality text images, and because of unusual letter-digit combinations like N30 or 30W; they may also be awkward with numbers written in words rather than digits.

Normalizing Text Descriptions

- The package produces multiple versions of the property descriptions in several steps, e.g.: *Retranscribing* characters misidentified by the OCR; *Rectifying* text where the user thinks the original deed was wrong; *Normalizing* abbreviated phrasing into more standard forms. For exmaple:
- The original transcription might say: "S 80 W 100 feet to the stream".
- The retranscribed description might say: "S 8 degrees W 100 feet to the stream", since the first 0 was probably a mistranscribed degree symbol.
- The rectified description might say: "N 8 degrees W 100 feet to the stream", if the user believes that the original text for this description was incorrectly written in the deed.
- The normalized description might say: "North 8 degrees West 100 feet to the stream", more fully written out for easier parsing later. In more common language, this boundary goes 8 degrees west of north for 100 feet.
- The software documents all transformations of the text.

Inferring Missing Data

- Older survey descriptions often omit some lengths. A deed might say only "north 8 degrees west to the stream".
- The package can infer missing lengths from three constraints: the *x-coordinate* of a boundary should end where it started; the *y-coordinate* of a boundary should end where it started; and the *area* of a parcel should agree with the area stated in the deed.
- The package will infer two missing lengths from the first two conditions, and will infer three missing lengths by using all three conditions. Future versions of the package are planned to infer angles also.

Mapping Individual Parcels

- The package's maps take into account various features of old deeds.
- Lengths in old deeds were often measured in chains (66 feet) and links (100ths of a chain).
- Angles in old deeds were typically measured relative to magnetic north, and magnetic north has varied substantially since 1800. This variation has been up to 12 degrees in New York, which is too large to neglect when lining up properties measured at different times.
- The package labels most borders with short descriptions of their angles and lengths, but for readability omits those descriptions on short edges.

Jigsawing Multiple Parcels

- The package also creates jigsaws of multiple parcels, based on which of their corners line up.
- The package will identify corners based on numbers (the fifth corner), directions (the northernmost point), or survey descriptions (the hemlock tree by the bridge).
- The package will also report the survey descriptions of the aligning vertices, to help verify the alignment.
- The jigsaws can include parcels described with different units and angles, and will include axes in both chains and feet as needed.

Functions

Splitting Strings

```
def split_at(string: str, separators: str, unseparators: str = [], default: str = 'Right')  
    > (str, str, str)
```

Split a string into three parts, with the separator as the middle part.

Return a tuple of three parts: (before, middle, after), where the middle part is the last occurrence of a separator.

The last argument says where to put the string if there is no separator.

The string could be 'then North 29 degrees West five chains to a point'.

Separators include '(North|South)', '(East|West)', 'to', '(feet|chains)'.

Unseparators are neighboring words indicating not to separate there, e.g.: 'the North' usually identifies a point as on the north side of something, 'North from' usually identifies a point as some distance from something, so they don't separate parts of a course like "then North 2 degrees West".

```
def split_deed(string: str, beginning: str, ending: str) > str
```

Get the string part from the given beginning to the given ending.

```
def split_parcel(string: str) > list[str]
```

Split a parcel into courses.

```
def split_course(course: str) > (str, str, str, str, str, str, str)
```

Split a course into seven parts for numerical extraction, e.g.:

("then", "North", "two degrees", "West", "four", "chains", "to a point") (zero, one, two, three, four, five, six)

Assume a single unit of length, e.g. 4.50 chains. A phrase like "four chains fifty links" should be transformed before using this function.

Conversions with Numbers

```
def wwords2wdigits(phrase: str) > str
```

Convert text with number-words to the same text with digits.

```
def wwords2float(phrase: str) > float
```

Extract the last number in a phrase and convert it to a float.

```
def words2degrees(phrase: str) > float
```

Convert a phrase to decimal degrees.

Retain only digits, periods and spaces.

Assume the first numbers are degrees, minutes, seconds.

```
def degrees2label(number: float) > str
```

Convert angle as degrees from x-axis into a bearing for a label.

```
def lonlatyear2magnet(lon: float = -77.01, lat: float = 38.89, year: int = 2023, month: int = 7,  
    day: int = 1) > (float, float)
```

Calculate a magnetic declination for a place and time.

```
def unit_dictionary() > dict
```

Dictionary of number of feet in a unit

```
def unit2feet(unit: str) > float
```

Convert a length unit into the corresponding number of feet.

Exclude 'foot' since it is more often 'the foot of the mountain'.

```
def squnits2acres(squnits: float, unit: str = 'feet') > float
```

Convert a number of square units with unit into a number of acres.

Optical Character Recognition

```
def pdf_to_text(pdffile: str, shortname: str, ocrtool: str = 'tesseract', overwrite: bool =  
    False) > str
```

Convert a pdf file to a text file, putting all the pages together.

By default use tesseract, otherwise use easyocr.

If there is already a text file with the results, use it.

Normalizing Text Descriptions

```
def replace_and_report(olds: list[str], news: list[str], string: str, stage: str = '') > (str,  
    list[str])
```

Replace old words in a string with new words, and report replacements.

Keep the same patterns of upper and lower case as before.

Mapping Individual Parcels

```
def do_segments_intersect(p: (float, float), q: (float, float), r: (float, float), s: (float, float)) > bool
```

Test if the segment from p to q intersects the segment from r to s.

Classes

```
class ParcelWords
```

Collect lists of words to be replaced and their replacements:

- transcriptold and transcriptnew for retranscriptions;
- normalold and normalnew for normalizations.

```
class Parcel (userinput: str, declination: float = 0, name: str = '', start: (float, float) = (0, 0), targetarea: float = 0, retranscriptions: list = [], rectifications: list = [], normalizations: list = [])
```

Convert the user input into coordinates and useful information.

Go through the following steps, in reverse-alphabetical order:

- 0 the user input is a survey description of a property from a deed;
- 1 the unbroken description removes line breaks;
- 2 the truncated description removes text not describing the bounds;
- 3 the retranscribed description fixes errors in transcription from OCR;
- 4 the rectified description fixes errors in the original deed;
- 5 the normalized description writes out some things in greater detail;
- 6 the matrix description puts the information in matrix format;
- 7 the inferred description inserts numbers for unknown lengths;
- 8 the inferred coordinates give courses in r, theta, x, y coordinates;
- 9 the cumulated coordinates give x, y coordinates for each corner;
- 10 the area is calculated from the cumulated coordinates.

Steps 0, 1, 2, 3, 4, 5, 7 all produce new versions of the text descriptions, mostly intended to mean the same thing.

Steps 6, 8, 9 produce complete numerical descriptions of the boundary.

Other methods are shown after the step that allows their computation.

Initialize a parcel from optional word lists and numerical data.

Cache blank values for the retranscribed and normalized descriptions These will be overridden later.

The inputs of retranscriptions, rectifications, normalizations specify transformations particular to this parcel, which are added to the generic lists of retranscriptions and normalizations. We do not have any generic legal errors to use as rectifications.

We can also initialize a parcel with: - a magnetic declination to be used with the course descriptions; - short name, which can be used in specifying jigsaw conditions; - a coordinate starting point, which is not usually known; - a target area, in case one is known from outside the given text.

Methods

```
def UnbrokenDescription(self) > (str, list[str])
```

Step 1: Remove line breaks.

```
def ExceptingOrReserving(self) > bool
```

Alert if the deed is "excepting and reserving" something.

```
def StatedArea(self) > float
```

Get the stated area in acres, which is often a useful constraint.

Deeds often state parcel areas outside the description of the bounds. So we compute it before truncating the description.

```
def TruncatedDescription(self) > (str, list[str])
```

Step 2: Cut text before and after the course descriptions.

```
def RetranscribedDescription(self) > (str, list[str])
```

Step 3: Fix characters misrecognized by the OCR transcription.

Cache as needed.

```
def RectifiedDescription(self) > (str, list[str])
```

Step 4: Manually fix errors in a parcel's legal description.

```
def NormalizedDescription(self) > (str, list[str])
```

Step 5: Standardize the description to get numbers more easily.

Cache as needed.

```
def Unit(self) > str
```

Use the unit that occurs most in the text.

```
def MatrixDescription(self) > list[str, str, str, str, str, str, str]
```

Step 6: Break down the description into rows of seven columns:

("then","North","two degrees","West","four","chains","to a point").

```
def CornerDescriptions(self) > list[str]
```

Describe the beginning and all of the corners.

```
def CornerDescription(self, index: float) > str
```

For an integer index (possibly 0), describe that numbered corner.

For an index between integers, describe the path between the corners.

```
def FourCoordinates(self, row: (str, str, str, str, str, str, str)) > (float, float,  
float, float)
```

Get r, theta, x, y from a row.

```
def IndexOf(self, word: str) > int
```

Say where a word first appears in the descriptions of corners.

```
def MatrixCoordinates(self) > list[float, float, float, float]
```

Give numbers for r, theta, x, y of each course.

This interprets "unknown1","unknown2","unknown3" as lengths of 1,2,3.

InferredCoordinates(self) has the numbers after inferring lengths.

```
def Test(self) > list[str]
```

Test the text before doing serious numerical computation.

```
def InferredLengths(self) > (float, float, float)
```

Find values for lengths unknown1, unknown2, unknown3 which:

- make the parcel end at the beginning,
- and optionally get the right target area for the parcel.

```
def InferredDescription(self) > (str, list[str])
```

Step 7: Infer a description with values for any unknown lengths.

The unknown lengths should be marked as unknown1, unknown2, unknown3. The word "unknown2" triggers inferences from lengths only. The word "unknown3" triggers inferences from lengths and area.

```
def OldLabelTexts(self) > list[str]
```

Provide labels for edges with text descriptions of how they run.

```
def InferredCoordinates(self) > list[float, float, float, float]
```

Step 8: Give r,theta,x,y for courses, using inferred lengths.

```
def NegativeSideLength(self) > bool
```

Warn about negative side lengths, usually from bad inferences.

```
def ReportOnWords(self) > list[str]
```

Report all edits made to the text of the description, by stage.

```
def ShortDescriptions(self) > list[str]
```

Describe courses briefly (e.g.: N39W 123.4') for labeling maps.

```
def CumulatedCoordinates(self, xys: list = []) > list[float, float]
```

Step 9: Calculate vertices by cumulating the course coordinates.

```
def Polygon(self)
```

Create a polygon for geometric computations from the vertices.

```
def LabeledPlot(self) > None
```

Plot a parcel, by default with arrows and short labels.

```
def SignedArea(self) > float
```

Calculate the signed area in acres, using the Polygon class.

The area is positive for counterclockwise descriptions. The area is negative for clockwise descriptions.

```
def Area(self) > float
```

Step 10: Calculate the area as a positive number of acres.

```
def ReportOnNumbers(self) > dict
```

Report numbers for the parcel, including errors and warnings.

```
def Report(self, type='short') > list
```

Provide a report on numbers, labeled plot, and report on words.

```
class Polygon (vertices: list[float, float])
```

This class has pure polygon geometry.

It does not refer to parcels, text, or units. It does refer to extreme points by N, E, S, W It calculates areas as square units, not as acres. It also plots and labels the individual polygon.

Initialize the polygon from its vertices, and nothing else.

Methods

```
def xError(self) > float
```

Get the x-distance between the polygon's start and end.

```
def yError(self) > float
```

Get the y-distance between the polygon's start and end.

```
def DistanceSq(self) > float
```

Get the distance² between the polygon's start and end.

```
def dError(self) > float
```

Get the distance between the polygon's start and end.

```
def Perimeter(self) > float
```

```
def SelfIntersecting(self) > bool
```

Test whether the polygon has self-intersections (it shouldn't).

Give the first self-intersecting pair or indexes, or return False.

```
def Extreme(self, short: float, output='coords') > float
```

Specify a point on a polygon by a direction or an index.

Return either coordinates or index of the point. Use a string input, expected to be at most three characters, either:

- a cardinal direction, e.g. N for the northernmost point;
- a diagonal direction, e.g. SW for the southwesternmost point;
- the number of a vertex;
- or a real number for a point partway down a side;
- but for reference by parcel descriptions, use the parcel's IndexOf.

```
def MidBox(self) > (float, float)
```

Find the middle of a polygon's bounding box.

This is a simple default position for a label.

```
def SignedArea(self) > float
```

Give the polygon's area in square units, not in acres.

Signed area is positive for counterclockwise descriptions. Signed area is negative for clockwise descriptions.

```
def Area(self) > float
```

Give the absolute value of the polygon's signed area.

```
def Concave(self, i: int) > bool
```

Test concavity of a vertex: is the polygon bigger without it?

```
def Plot(self, arrows: bool = True, labels: list[str] = [], lims: list[float, float] = [], rotate: bool = False) > None
```

Plot a polygon or part of it, with arrows and labels on sides.

```
class Jigsaw (parcels_or_polygons: list, conditions: list[str] = [], units: list[str] = [], names: list[str] = [], labels: list[str] = [])
```

Create a jigsaw map of parcels or polygons by aligning corners.

Also report the parcel descriptions at those aligned corners. The alignments can say "A's NW is B's NE", or use other descriptions. Alignments after the first should introduce only one new polygon at a time.

Initialize the jigsaw from parcels or polygons, and other things.

The first initialization input is a list of parcels or polygons. In either case we get a list of polygons and a list of parcels. If the input list has parcels, we create a list of their polygons. If the input list has polygons, we create an empty list of parcels.

The other initialization inputs are: - the conditions of polygon alignments - the units for each polygon. - the names for each polygon, for use in the condition. - the labels for each polygon, for using in the plotting The units, names or labels can all be inferred from parcels.

Methods

```
def Factors(self) > list[float]
```

If the parcels have both feet and chains, multiply chains by 66.

```
def ConditionsData(self) > list[dict]
```

Get coordinates, indices, and explanations for the conditions.

Mostly use polygons, with parcels only for survey descriptions.

```
def Starts(self) > list[float, float]
```

Find starts for the polygons that satisfy the conditions.

```
def Corners(self) > (float, float, float, float)
```

Identify N, E, S, W extremes among the polygons using the Starts.

```
def Plot(self) > None
```

Plot the jigsaw from scratch, without using polygon plots.

This allows a mix of two units, e.g. feet and chains. This does not allow arrows or edge labels.

```
def Report(self) > list[str]
```

Show the plot and print explanations about which points overlap.

Index

Optical Character Recognition
Normalizing Text Descriptions
Inferring Missing Data
Mapping Individual Parcels
Jigsawing Multiple Parcels

Functions

split_at
split_deed
split_parcel
split_course
wwords2wdigits
wwords2float
words2degrees
degrees2label
lonlatyear2magnet
unit_dictionary
unit2feet
squnits2acres
pdf_to_text
replace_and_report
do_segments_intersect

Classes

ParcelWords

Parcel

UnbrokenDescription
ExceptingOrReserving
StatedArea
TruncatedDescription
RetranscribedDescription
RectifiedDescription
NormalizedDescription
Unit

MatrixDescription
CornerDescriptions
CornerDescription
FourCoordinates
IndexOf
MatrixCoordinates
Test
InferredLengths
InferredDescription
OldLabelTexts
InferredCoordinates
NegativeSideLength
ReportOnWords
ShortDescriptions
CumulatedCoordinates
Polygon
LabeledPlot
SignedArea
Area
ReportOnNumbers
Report

Polygon

xError
yError
DistanceSq
dError
Perimeter
SelfIntersecting
Extreme
MidBox
SignedArea
Area
Concave
Plot

Jigsaw

Factors
ConditionsData
Starts
Corners

