CS1010S Programming Methodology

Lecture 9

Generic Operations

24 Oct 2018

Today's Agenda

- Python Dictionary
- Recap: Complex-Arithmetic Package
- Tagging Data
- Implementing Generic Operators
  - Dispatch on Type
  - Data-Directed Programming
  - Message Passing

Python Dictionary

- A dictionary is a table of key-value pairs (aka associative array)
  - enclosed in curly brackets
  - allows retrieval by key
  - keys are unique within a dictionary
  - keys must be of an immutable data type such as strings, numbers, or tuples
  - values can be of any type

Python Dictionary

- To create a dictionary:

```
weather = {
    'wind': 0,  # key: value
    'description': 'cloudy',
    'temp': {2: 26.8, 14: 31.1} # nested
}
```

- We can also use the built-in function `dict` to create a dictionary:

```
weather = dict(wind = 0,  # key = value
               description = 'cloudy',
               temp = {2:26.8, 14:31.1})
```

Python Dictionary

```
>>> weather = {
    'wind': 0,   # key: value
    'description': 'cloudy',
    'temp': {2: 26.8, 14: 31.1} # nested
}

>>> weather['temp’]   # retrieve value using key
{2: 26.8, 14: 31.1}

>>> 'wind' in weather # check if key exists
True

>>> 0 in weather # no such key
False
```

Python Dictionary

```
>>> weather = {
    'wind': 0,  # key: value
    'description': 'cloudy',
    'temp': {2: 26.8, 14: 31.1} # nested
}

>>> weather['wind'] = 1  # update an existing entry

>>> weather['is_nice'] = True # add a new entry

>>> del weather['temp']  # delete an entry

>>> weather
{'wind': 1, 'description': 'cloudy', 'is_nice': True}
```

Python Dictionary

```
>>> list(weather.keys())
['wind', 'description', 'is_nice']

>>> list(weather.values())
[1, 'cloudy', True]

>>> len(weather)  # no. of key-value pairs
3

>>> weather.clear() # delete all entries

>>> weather
{} # empty dictionary
```

Python Dictionary

```
>>> weather = { 'wind': 1, 'description': 'cloudy',
                'is_nice': True }

>>> for key in weather:
        print(weather[key])
1
cloudy
True

>>> for key, value in weather.items():
        print(key, ':', value)
wind : 1
description : cloudy
is_nice : True
```

Today's Agenda

- Python Dictionary
- Recap: Complex-Arithmetic Package
- Tagging Data
- Implementing Generic Operators
  - Dispatch on Type
  - Data-Directed Programming
  - Message Passing

Abstraction barrier

PREVIOUS LECTURE

Programs that use complex numbers

(use given functions)

`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package

Rectangular representation	Polar representation

Rectangular Rep.

PREVIOUS LECTURE

```
import math
def make_from_real_imag(x, y):
    return (x, y) # internal representation
def real_part(z):
    return z[0]
def imag_part(z):
    return z[1]
def magnitude(z):
    return math.hypot(real_part(z), imag_part(z))
def angle(z):
    return math.atan(imag_part(z)/real_part(z))
def make_from_mag_ang(r, a):
    return (r * math.cos(a), r * math.sin(a))
```

Polar Rep.

PREVIOUS LECTURE

```python
import math
def make_from_mag_ang(r, a):
    return (r, a) # internal representation
def magnitude(z):
    return z[0]
def angle(z):
    return z[1]
def real_part(z):
    return magnitude(z) * math.cos(angle(z))
def imag_part(z):
    return magnitude(z) * math.sin(angle(z))
def make_from_real_imag(x, y):
    return (math.hypot(x, y), math.atan(y/x))
```

Complex Number Operations

```
def add_complex(z1, z2):
    return make_from_real_imag(real_part(z1) + real_part(z2),
                               imag_part(z1) + imag_part(z2))


def mul_complex(z1, z2):
    return make_from_mag_ang(magnitude(z1) * magnitude(z2),
                             angle(z1) + angle(z2))


def print_complex(z): # print x+yi
    print(str(real_part(z)) + '+' + str(imag_part(z)) + 'i')


# sub_complex() and div_complex() functions skipped
```

Code in Action

PREVIOUS LECTURE

```
>>> from complex_rectanglar_rep import *
>>> a = make_from_real_imag(1, 2)
>>> b = make_from_real_imag(1, 1)
>>> print_complex(add_complex(a, b))
2+3i
```

Using the functions from rectangular rep.

```
>>> from complex_polar_rep import *
>>> a = make_from_real_imag(1, 2)
>>> b = make_from_real_imag(1, 1)
>>> print_complex(add_complex(a, b))
2.0000000000000004+3.0i
```

Using the functions from polar rep.

Multiple Representations

- Typically in large software projects, multiple representations co-exist.
  - Because large projects have long lifetime, and project requirements change over time.
  - Because no single representation is suitable for every purpose.
  - Because programmers work independently and develop their own representations for the same thing.
  - etc.

Issues with Co-existing Rep.

- Matching representations to operations
  - e.g. rectangular rep. must be given to rectangular functions; polar rep. cannot be given to rectangular functions, or vice versa

- Name conflicts
  - Both reps have functions with identical names, e.g. `real_part`, `magnitude`, etc.

Matching Rep. to Operations

- Issue: matching representations to operations.

- Solution: tagged data
  - Each representation is given a tag to explicitly indicate the representation (type).
  - Use tuple: `(tag, content)`
    - e.g. `('rectangular', content)`
    - e.g. `('polar', content)`

Today's Agenda

- Python Dictionary
- Recap: Complex-Arithmetic Package
- Tagging Data
- Implementing Generic Operators
  - Dispatch on Type
  - Data-Directed Programming
  - Message Passing

Tagging

```
def attach_tag(type_tag, contents):
    return (type_tag, contents)

def type_tag(datum):
    if type(datum) == tuple and len(datum) == 2:
        return datum[0]
    else:
        raise Exception('Bad tagged datum --\
                        type_tag' + str(datum))
```

Exception: Signals something bad has happened

Tagging

```
def contents(datum):
    if type(datum) == tuple and len(datum) == 2:
        return datum[1]
    else:
        raise Exception('Bad tagged datum --\
                         contents' + str(datum))
```

Tagged Data

- To check for tag:

```
def is_rectangular(z):
    return type_tag(z) == 'rectangular'

def is_polar(z):
    return type_tag(z) == 'polar'
```

Revised Rectangular Rep.

```
import math
def make_from_real_imag(x, y):
    return attach_tag( 'rectangular', (x, y) )
def real_part(z): # selector: return real part
    return z[0]
def imag_part(z): # selector: return imaginary part
    return z[1]
def magnitude(z):
    return math.hypot(real_part(z), imag_part(z))
def angle(z):
    return math.atan(imag_part(z)/real_part(z))
def make_from_mag_ang(r, a):
    return attach_tag( 'rectangular', ... )
```

Revised Polar Rep.

```
import math
def make_from_mag_ang(r, a):
    return attach_tag( 'polar', (r, a) )
def magnitude(z): # selector
    return z[0]
def angle(z): # selector
    return z[1]
def real_part(z):
    return magnitude(z) * math.cos(angle(z))
def imag_part(z):
    return magnitude(z) * math.sin(angle(z))
def make_from_real_imag(x, y):
    return attach_tag( 'polar', ... )
```

Resolving Name Conflicts

- Issue: both representations have the same function names.
- One solution: impose naming convention – every function name ends with a tag

```
# e.g. in rectangular representation:
def make_from_real_imag_rectangular(z):
    return attach_tag('rectangular', (x, y))
def real_part_rectangular(z):
    return z[0]
# other functions skipped for brevity
```

Who should Check Type?

```
def add_complex(z1, z2):
    if is_rectangular(z1):
        real1 = real_part_rectangular(contents(z1))
        imag1 = imag_part_rectangular(contents(z1))
    else: # data in polar representation
        real1 = real_part_polar(contents(z1))
        imag1 = imag_part_polar(contents(z1))
    if is_rectangular(z2):
        real2 = real_part_rectangular(contents(z2))
        imag2 = imag_part_rectangular(contents(z2))
    else:
        real2 = real_part_polar(contents(z2))
        imag2 = imag_part_polar(contents(z2))
    return make_from_real_imag_rectangular(real1 + real2,
                                           imag1 + imag2)
```

User has to check for rep. and call appropriate functions

Whither the Future?

- What if one or more of the following happen in the future?
  - Rectangular code is removed. Only polar representation is left.
  - A new representation for complex number is installed, and co-exists with current representations.

- How to minimize the effects of the above changes?

Today's Agenda

- Python Dictionary
- Recap: Complex-Arithmetic Package
- Tagging Data
- Implementing Generic Operators
  - Dispatch on Type
  - Data-Directed Programming
  - Message Passing

Abstraction barrier

Programs that use complex numbers

`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package

Rectangular representation	Polar representation

New Abstraction Layer

Programs that use complex numbers

`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package

Generic Operators ← new layer

Rectangular representation	Polar representation

Generic Operators

- Create another layer of abstraction.
  - A set of generic functions.
  - To shield user from the complexity of managing multiple representations.
  - ADT designer, user, or project leader can create this layer.

Generic Operators

- Example:

User uses this generic operator.

```
def real_part(z):
    if is_rectangular(z):
        return real_part_rectangular(contents(z))
    elif is_polar(z):
        return real_part_polar(contents(z))
    else:
        raise Exception('Unknown type -- real_part' + z)
# generic operators imag_part, magnitude... etc. skipped
```

- Type checking, tag removal are hidden from user.

User's Code

```
def add_complex(z1, z2):
    return make_from_real_imag(real_part(z1) + real_part(z2),
                               imag_part(z1) + imag_part(z2))
# sub_complex, mul_complex, div_complex functions skipped
```

- User’s code simplified.
- Selectors (`real_part`, `imag_part`) are generic operators.

Strategy #1: Dispatching on type

- This strategy of providing generic operators based on "checking the data type and calling the appropriate function" is called dispatching on type.

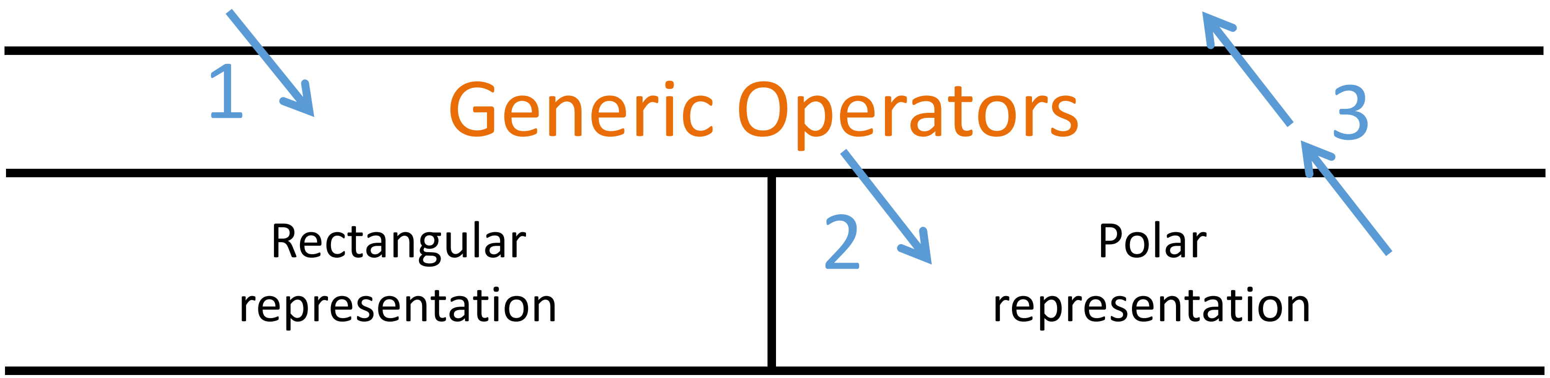
- Generic operators work on data that could take on *multiple forms* (*polymorphic data*).

Moving Across Barriers

1. User calls generic operator to work on polymorphic data.
2. Generic operator dispatches on type. Data is stripped of tag when going down.
3. Returned data may be tagged with type when going up.

```
add_complex, sub_complex, mul_complex, div_complex
```

Complex Numbers Package



Thinking

- In the current design of generic operators:
  - Generic operators need to know all the types (rectangular, polar, etc.) available.
  - Adding a new type means changing all operators to dispatch correctly.
  - Does not resolve name conflict (if the same function name, e.g. real_part is used in different representations).

Strategy #2: Data-directed Programming

- A better design: how about using a table and doing a table lookup?

- Generic operators look at the tag on data and find the correct operation from table.
  - Address problem of naming conflicts
  - Allows easy extension: just add more entries to the table!

Table of Operations

- In creating generic operators, we are really selecting the appropriate lower-level function based on operation and type, as summarized in table `procs` below.

Table: procs

		Types	
		Polar	Rectangular
Operations	real_part	real_part	real_part
	imag_part	imag_part	imag_part
	magnitude	magnitude	magnitude
	angle	angle	angle

Table Manipulation

- We can implement the table `procs` as a Python dictionary.

```
# installs <value> in table, indexed by <op> and <type>
def put(op, type, value):
    if op not in procs:
        procs[op] = {} # empty dictionary
    procs[op][type] = value

# looks up <op>, <type> entry in table and
# returns the value found there
def get(op, type):
    return procs[op][type]
```

Table Manipulation

	Types	
Operations	Polar	Rectangular
real_part	real_part	real_part
imag_part	imag_part	imag_part
magnitude	magnitude	magnitude
angle	angle	angle

Key	Value
real_part	
...	

Key	Value
polar	real_part
rectangular	real_part

Key	Value
polar	...
rectangular	...

Installing Rect. Package - 1

```
def install_rectangular_package():
    def make_from_real_imag(x, y):
        return attach_tag('rectangular', (x, y))
    def real_part(z):
        return z[0]
    def imag_part(z):
        return z[1]
    def magnitude(z):
        return math.hypot(real_part(z), imag_part(z))
    def angle(z):
        return math.atan(image_part(z)/real_part(z))
    # to continue next page
```

Installing Rect. Package - 2

```
def make_from_mag_ang(r, a):
    return make_from_real_imag(r*math.cos(a),
                               r*math.sin(a))
# insert into table
put('real_part', ('rectangular',), real_part)
put('imag_part', ('rectangular',), imag_part)
put('magnitude', ('rectangular',), magnitude)
put('angle', ('rectangular',), angle)
put('make_from_real_imag', 'rectangular',
     make_from_real_imag)
put('make_from_mag_ang', 'rectangular',
    make_from_mag_ang)
return 'done'
```

Installing Polar Package - 1

```
def install_polar_package():
    def make_from_mag_ang(r, a):
        return attach_tag('polar', (r, a))
    def magnitude(z):
        return z[0]
    def angle(z):
        return z[1]
    def real_part(z):
        return magnitude(z) * math.cos(angle(z))
    def imag_part(z):
        return magnitude(z) * math.sin(angle(z))
    # to continue next page
```

Installing Polar Package - 2

```
def make_from_real_imag(x, y):
    return make_from_mag_ang(math.hypot(x, y),
                             math.atan(y/x))
# insert into table
put('real_part', ('polar',), real_part)
put('imag_part', ('polar',), imag_part)
put('magnitude', ('polar',), magnitude)
put('angle', ('polar',), angle)
put('make_from_real_imag', 'polar',
     make_from_real_imag)
put('make_from_mag_ang', 'polar',
    make_from_mag_ang)
return 'done'
```

Resulting Table

op

type of data

function to return

```
>>> procs
{'real_part':
    {('polar',): <function install_polar_package...real_part...>,
     ('rectangular',): <func...rectangular_package...real_part...>},
 'imag_part':
    {('polar',): <function install_polar_package...imag_part...>,
     ('rectangular',): <func...rectangular_package...imag_part...>},
 'magnitude':
    {('polar',): <function install_polar_package...magnitude...>,
     ('rectangular',): <func...rectangular_package...magnitude...>},
 'angle':
    {('polar',): <function install_polar_package...angle...>,
     ('rectangular',): <func...rectangular_package...angle...>},
  ...
}
```

Observation

- No name conflicts even if different representations use the same function names!
  - Because all function names are internal (local) to the installer function.
  - Function names live in separate *name space*.
  - Thus, each programmer can use identical names.

- Installer defines all operators and places them in the table procs according to operation and type.

Generic Operators

- Generic operators in strategy #2:

```
def real_part(z):
    return get('real_part', type_tag(z))(contents(z))
def imag_part(z):
    return get('imag_part', type_tag(z))(contents(z))
def magnitude(z):
    return get('magnitude', type_tag(z))(contents(z))
def angle(z):
    return get('angle', type_tag(z))(contents(z))
# generic constructors skipped for brevity
```

Generic Operators

- To compare: generic operators in strategy #1:

```
def real_part(z):
    if is_rectangular(z):
        return real_part_rectangular(contents(z))
    elif is_polar(z):
        return real_part_polar(contents(z))
    else:
        raise Exception('Unknown type -- real_part' + z)
# other generic operators skipped for brevity
```

The * Notation

- Next, we will learn a new syntax, `*args`, which represents any number of arguments.
  - Useful in your missions!

```
def f(x, y, *args):
    <body>
```

- Function `f` can be called with 2, or more arguments. For example,

```
>>> f(1, 2, 3, 4)
# x is 1, y is 2
# args is tuple (3, 4)
```

Example

```python
def multiply(x, y):
    print(x*y)
multiple(2, 10)      # prints 20
multiple(2, 3, 10)  # fail to run
```

- To multiple any number of integers:

```python
def multiply(*args):
    res = 1
    for num in args:
        res = res * num
    print(res)
multiple(2, 3, 10)  # prints 60
```

The * Notation

- You can also call a function for which you don't know in advance how many arguments there will be using *.

```
def funky(op, args):
    return op(*args) # call a function with *args

funky( lambda x: x*x, (2,) )       # returns 2*2 = 4
funky( lambda x, y: x+y, (2, 1) ) # returns 2+1 = 3
```

Making Generic Operators

```python
def apply_generic(op, *args):
    type_tags = tuple( map(type_tag, args) )
    proc = get(op, type_tags)
    return proc( *map(contents, args) )
```

```python
# defined on page 19 and repeat here
def type_tag(datum):
    if type(datum) == tuple and len(datum) == 2:
        return datum[0]
    else:
        raise Exception('Bad tagged datum --\
                                 type_tag' + str(datum))
```

Example

```
def apply_generic(op, *args):
    type_tags = tuple( map(type_tag, args) )
    proc = get(op, type_tags)
    return proc( *map(contents, args) )

apply_generic('real_part', z)
# op is 'real_part'
# args is (z,), e.g. ( ('polar', (3, 4)), )
# type_tags = ('polar',)
# proc = get('real_part', ('polar',))
#      = <function install_polar_package.<locals>.real_part...
# invoke real_part(contents(z)) of polar rep.
```

Notes

- `apply_generic()` retrieves appropriate function `proc()` from table procs based on <op> and <type>.

- Then applies `proc()` to the list of arguments.

- This style of using a table to dispatch is called data-directed programming.

Generic Operators

```
def apply_generic(op, *args):
    type_tags = tuple( map(type_tag, args) )
    proc = get(op, type_tags)
    return proc( *map(contents, args) )
def real_part(z):
    return apply_generic('real_part', z)
def imag_part(z):
    return apply_generic('imag_part', z)
def magnitude(z):
    return apply_generic('magnitude', z)
def angle(z):
    return apply_generic('angle', z)
```

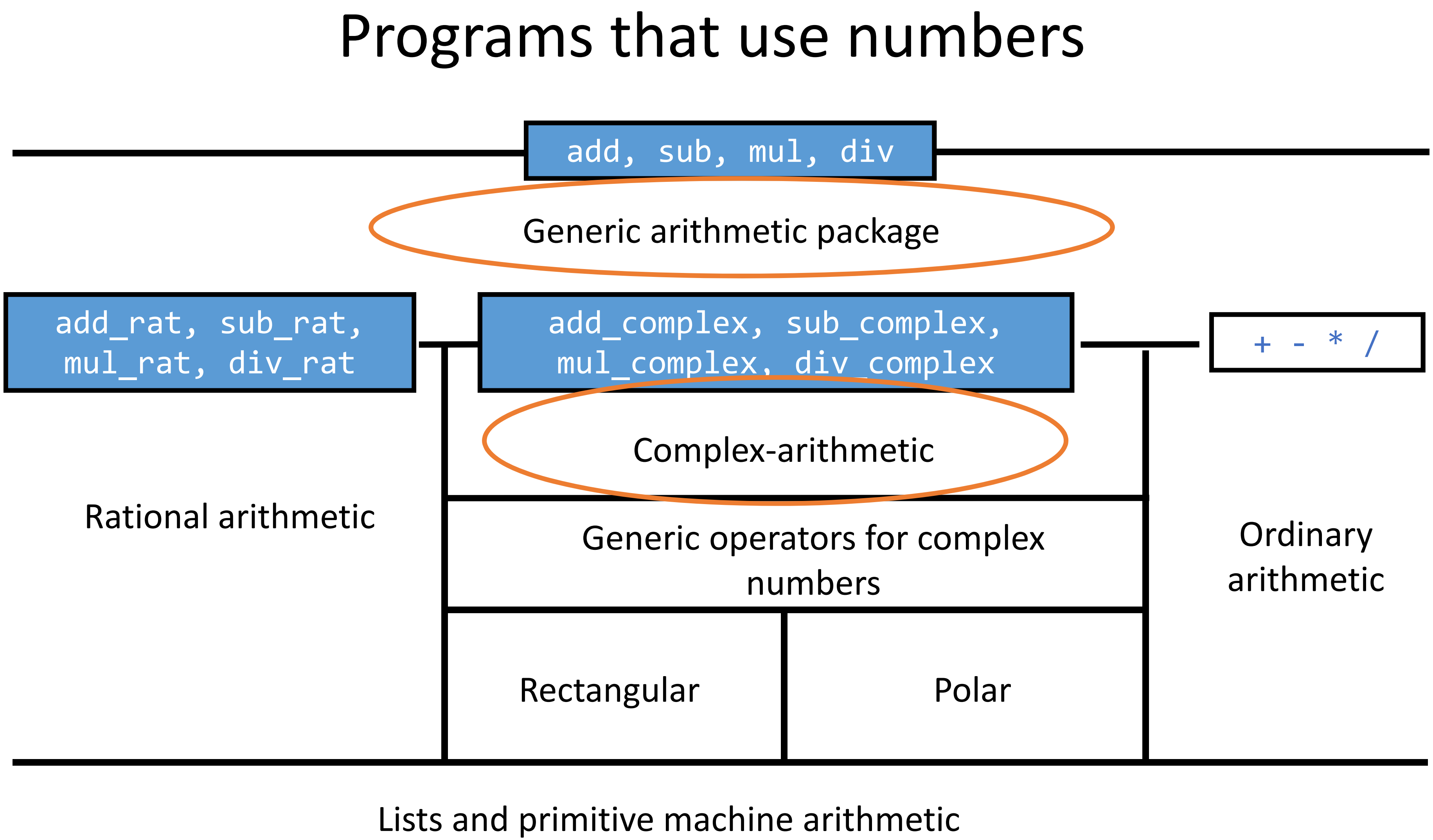
Generic Operators

```
# Constructors
def make_from_real_imag_rectangular(x, y):
    return get('make_from_real_imag', 'rectangular')(x, y)

def make_from_mag_ang_rectangular(x, y):
    return get('make_from_mag_ang', 'rectangular')(x, y)

def make_from_real_imag_polar(x, y):
    return get('make_from_real_imag', 'polar')(x, y)

def make_from_mag_ang_polar(x, y):
    return get('make_from_mag_ang', 'polar')(x, y)
```

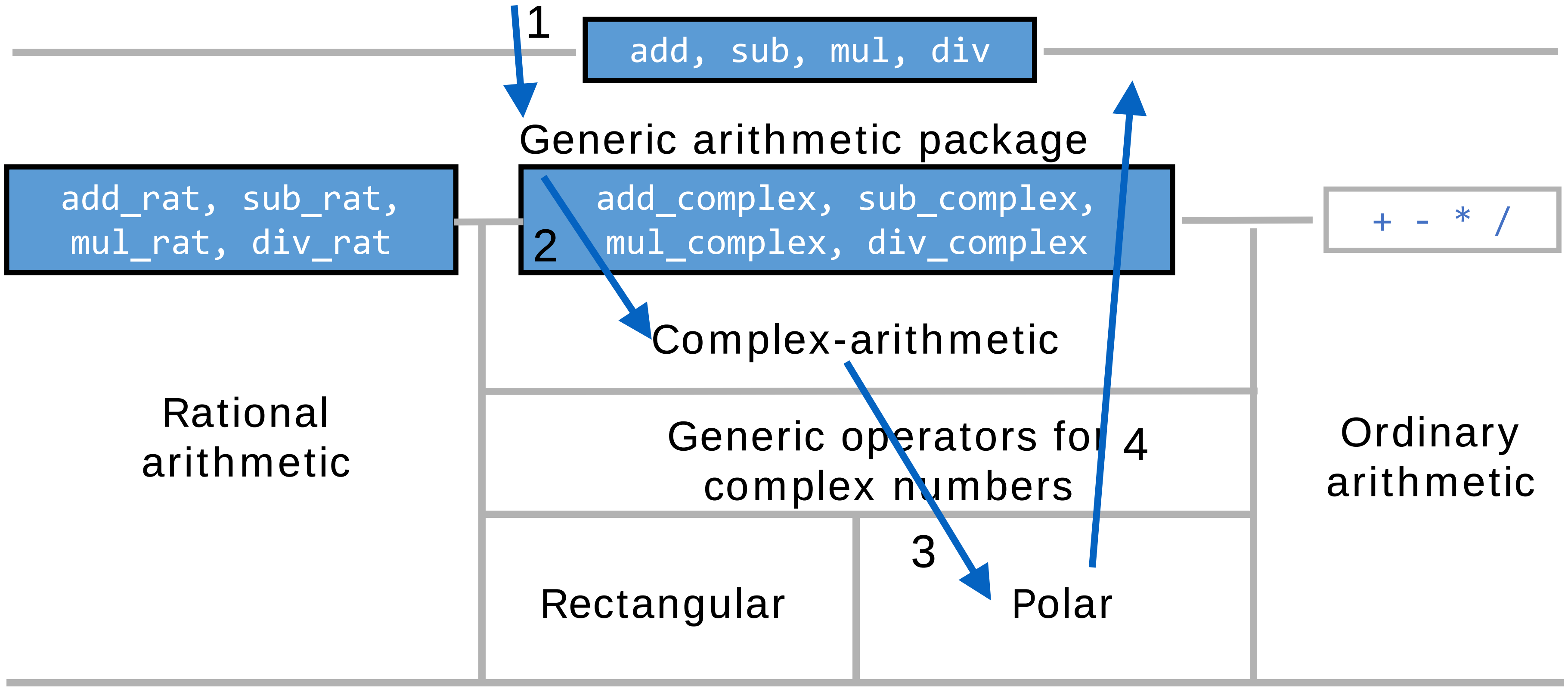
Generic Arithmetic

- We can push this idea further!
- We can use generic operators to handle *completely different data*, not just different representations of the same data.
- Let's build a generic arithmetic package.
  - Work with rational numbers, complex numbers, ordinary numbers, even polynomials!
  - Provide generic operators: add, sub, mul, div

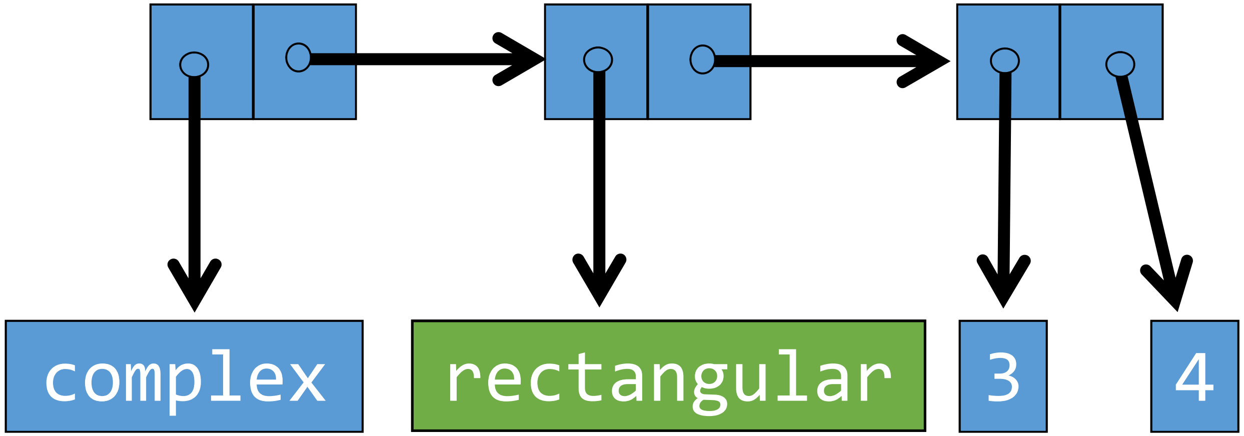
Generic Arithmetic



- User calls generic arithmetic operators to work on polymorphic numbers.
- Generic operator dispatches on type. Data is stripped of tag going down, e.g. 'complex'
- Complex package calls generic operators for rect/polar forms. Tag is stripped for data going down.
- Returned data tagged with types going up, e.g. 'polar', and 'complex'.



Tags upon Tags



Strategy # 3: Message Passing

- Previous strategies viewed functions as "intelligent":
  - They dispatch according to type of data.

- In *message passing*, it is the data that is "intelligent":
  - They know how to act on themselves.
  - You just "tell" data what you want.

Message Passing

```
def make_from_real_imag(x, y):
    def dispatch(op):
        if op == 'real_part':
            return x
        elif op == 'imag_part':
            return y
        elif op == 'magnitude':
            return math.hypot(x, y)
        elif op == 'angle':
            return math.atan(y / x)
    return dispatch
```

Message Passing

- "Data" is the function `dispatch`.
    - dispatch() accepts a "message" op, and performs the necessary action according to op.

```
def real_part(z):
    return z('real_part')
def imag_part(z):
    return z('imag_part')
```

```
>>> z = make_from_real_imag(1, 2)
>>> real_part(z)
1
```

Message Passing

- This idea of "passing a message" to the data and letting the data do the job is the basis of object-oriented programming.
  - Data are objects
  - Functions are actions that objects perform.
  - Objects "talk" to other objects by passing messages.
  - More will be introduced in recitation.

Summary

- Challenges of managing multiple representations.
  - Matching data types to operations
  - Resolving name conflicts
  - Managing changes/future modifications
- Abstraction layer between the underlying representations and the user

Summary

- You have now seen 3 strategies for creating generic operators:
  - Dispatching on type
  - Data-directed programming
  - Message passing

- Abstraction layer between the underlying representations and the user