Announcements

- Please submit exam conflict forms by Thursday (see Piazza @318).
- Drop deadline today.
- Ask questions on the Piazza thread for today's lecture (Piazza @794).

Containers

- So far, we've looked at numbers and functions.
- Numbers are a kind of atomic data: we don't really think of them as having parts, but rather deal with them as wholes.
- But how are we to handle containers of data: data that consists of other data?
- In fact, we can already do that with what we have!

Simple Pairs (Take 1)

- Let's suppose I'd like to be able to represent a pair of values as a single data value.
- What should I be able to do with this type?
  - Construct new pairs from two values.
  - Select one or the other of the two values used to construct the pair.
- So our goal might be summarized with these specifications:
  ```python
  def pair(a, b):
      """Return a value that represents the ordered pair of values (A, B)."""

  def left(p):
      """Assuming that P was created by pair(x, y), return the value x."""

  def right(p):
      """Assuming that P was created by pair(x, y), return the value y."""
  ```
- With what we have, how can we make this work?
Fun Side Trip: Pairs of Integers

• One trick works with pairs of integers. Here, let’s just stick with non-negative integers.
• (Kurt Gödel used this in a very famous theorem.)
• Define

  ```python
def pair(a, b):
      """Return a value that represents the ordered pair of non-negative integer values (A, B)."""
      return 2**a * 3**b

def left(p):
    return multiplicity(2, p)

def right(p):
    return multiplicity(3, p)

def multiplicity(factor, n):
    """Assuming FACTOR and N are integers with FACTOR > 1, return the number of times N may be evenly divided by FACTOR."""
    # Implementation left to the reader

• Of course, this representation is not particularly practical!

Another Representation

• Instead of integers, we can use functions for our representation:

  ```python
def pair(a, b):
      """Return a value that represents the ordered pair of values (A, B).""
      return ??
def left(p):
    """Assuming that P was created by pair(x, y), return the value x.""
    return ??
def right(p):
    """Assuming that P was created by pair(x, y), return the value y.""
    return ??

def multiplicity(factor, n):
    """Assuming FACTOR and N are integers with FACTOR > 1, return the number of times N may be evenly divided by FACTOR."""
    # Implementation left to the reader

• Of course, this representation is not particularly practical!

Another Representation

• Instead of integers, we can use functions for our representation:

  ```python
def pair(a, b):
      """Return a value that represents the ordered pair of values (A, B).""
      return ??
def left(p):
    """Assuming that P was created by pair(x, y), return the value x.""
    return ??
def right(p):
    """Assuming that P was created by pair(x, y), return the value y.""
    return ??

def multiplicity(factor, n):
    """Assuming FACTOR and N are integers with FACTOR > 1, return the number of times N may be evenly divided by FACTOR."""
    # Implementation left to the reader
Another Representation

- Instead of integers, we can use functions for our representation:
  
  ```python
  def pair(a, b):
      """Return a value that represents the ordered pair of values (A, B).""
      return lambda which: a if which == 0 else b
  
  def left(p):
      """Assuming that P was created by pair(x, y), return the value x.""
      return p(0)
  
  def right(p):
      """Assuming that P was created by pair(x, y), return the value y.""
      return p(1)
  
  Feature 1: This is the same spec as the previous one (aside from the restriction to integers). Any program using pairs of integers will work without change with this new representation.
  
  Feature 2: With just these functions, these values are immutable: to change either part of the pair, one must create a new pair.
  
```

Adding Mutability

- What if we want to expand the spec a bit to make it possible to set one or the other members of a given pair? That is, add the following:
  
  ```python
  def set_left(p, v):
      """Given that P represents the pair (x, y), cause P to represent (v, y), returning None.""
      return p(2, v)
  
  def set_right(p, v):
      """Given that P represents the pair (x, y), cause P to represent (x, v), returning None.""
      return p(3, v)
  
  Attempt #1
  
  - Well, let's just try an extension of the same idea.
  
  - First, define the new methods as follows
    ```python
    def set_left(p, v):
      p(2, v)
    
    def set_right(p, v):
      p(3, v)
    
    - Next, re-implement pair:
      ```
      ```python
      def pair(a, b):
          def pair_func(which, v=None):
              if which == 0:
                  return a
              elif which == 1:
                  return b
              elif which == 2:
                  # What goes wrong?
                  a = v
              else:
                  b = v
          return pair_func
      ```
      ```
  
  Attempt #1
  
  - Well, let's just try an extension of the same idea.
  
  - First, define the new methods as follows
    ```python
    def set_left(p, v):
      p(2, v)
    
    def set_right(p, v):
      p(3, v)
    
    - Next, re-implement pair:
      ```
      ```python
      def pair(a, b):
          def pair_func(which, v=None):
              if which == 0:
                  return a  # Not the right a and b!
              elif which == 1:
                  return b
              elif which == 2:
                  # What goes wrong?
                  a = v
              else:
                  b = v
          return pair_func
      ```
Nonlocal

- Assignment in Python usually creates or sets a local variable in the currently executing environment frame.
- But that's useless in the attempted implementation (see the test_bad_pair function in 10.py).
- We need instead to indicate that we actually want to set the variables a and b introduced outside the pair_func function in the enclosing (parent) function's (pair's) frame.
- The declaration
  ```python
  nonlocal var1, var2, ...
  ```
  means "assignment to any of the variables var_i in the current frame actually assigns to those variables in its parent's frame, grandparent's frame, etc. (not including the global frame)."
- Furthermore, those variables must already exists in one of these ancestor frames.

Attempt #2

- So the fix is to rewrite pair as follows:
  ```python
  def pair(a, b):
    # <--- The nonlocal a and b.
    def pair_func(which, v=None):
      nonlocal a, b  # a and b refer to variables in pair's header.
      if which == 0:
        return a
      elif which == 1:
        return b
      elif which == 2:
        a = v
      else:
        b = v
      return pair_func
  ```

Getting Real

- It's interesting to see the function-base implementation of pair because it shows how one can get by with very few basic features in a language.
- You'll see something like this employed in Javascript programs, in fact, although for somewhat different purposes.
- However, as a way to represent data structures such as tuples, it is rather inefficient.
- Therefore, Python provides other, more customized representation and syntax for these things.

Sequences

- The term sequence refers generally to a data structure consisting of an indexed collection of values, which we'll generally call elements.
- That is, there is a first, second, third value (which CS types call #0, #1, #2, etc.)
- A sequence may be finite (with a length) or infinite.
- It may be mutable (elements can change) or immutable.
- It may be indexable: its elements may be accessed via selection by their indices.
- It may be iterable: its values may be accessed sequentially from first to last.
Python's Sequences

• There are several different kinds of sequence embodied in the standard Python types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Elements</th>
<th>Indexable?</th>
<th>Mutable?</th>
<th>Iterable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuple</td>
<td>Any type</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>list</td>
<td>Any type</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>str ()</td>
<td>str (!)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>range</td>
<td>integer</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>iterator</td>
<td>Any type</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• We'll take up iterators and generators in a later lecture.

• Python goes to some lengths to provide a uniform interface to all the various sequence types, as well as to its other collection types, including sets and dictionaries.

String Literals

• For convenience and readability, strings have a fairly rich variety of literals.

  'Single-quoted strings may contain "double-quoted strings"'

  "Double-quoted strings may contain 'single-quoted strings"

  """Triple double quotes allow 'this', "this", and ""this"", as well as newline characters"

  '''Triple single quotes allow "this", 'this', and '''this'', as well as newline characters'''

• Escape sequences allow all these types of string to contain quotes, newlines, backslashes, and other symbols, even non-ASCII characters.

  >>> "A test of\nescapes\."" "A test of\nescapes\.""

  >>> "Some unicode: \u0395\u1f55\u03c1\u03b7\u03ba\u2764"
  Some unicode: Ερήμος

  >>> r"In raw strings (starting with 'r'), \escapes are not replaced"" In raw strings (starting with 'r'), \escapes are not replaced

Selection and Slicing

• Selection refers to extracting elements by their index.

• Slicing refers to extracting subsequences.

• These work uniformly across sequence types.

  t = (2, 0, 9, 10, 11)   # Tuple
  L = [2, 0, 9, 10, 11]   # List
  R = range(2, 13)       # Integers 2-12.
  E = range(2, 13, 2)    # Even integers 2-12.
  S = "Hello, world!"   # Strings (sequences of characters)

  t[-1] == t[len(t)-1] == 11
  S[1] == "e"   # Each element of a string is a one-element string.

  t[1:4] == (t[1], t[2], t[3]) == (0, 9, 10),
  t[2:] == t[2:len(t)] == (9, 10, 11)
  t[:2] == t[0:len(t):2] == (2, 9, 11), t[::1] == (11, 10, 9, 0, 2)
  S[1:2] == S[1] == "e"
Sequence Combination and Conversion

- Sequence types can be converted into each other where needed:
  - `list((1, 2, 3)) == [1, 2, 3]`
  - `tuple([1, 2, 3]) == (1, 2, 3)`
  - `list(range(2, 10, 2)) == [2, 4, 6, 8]`
  - `list("ABCD") == ['A', 'B', 'C', 'D']`

- One can construct certain sequences (tuples, lists, strings) by concatenating smaller ones:
  - `A = [1, 2, 3, 4]`
  - `B = [7, 8, 9]`
  - `A + B == [1, 2, 3, 4, 7, 8, 9]`
  - `(1, 2, 3, 4) + (7, 8, 9) == (1, 2, 3, 4, 7, 8, 9)`
  - "Hello," + " " + "world" == "Hello, world"
  - `(1, 2, 3, 4) + 3` ERROR (why?)

Sequence Iteration: For Loops

- Using selection and the `len` function on sequences (which gives their length), we can operate on each element of a sequence.

- However, we can write more compact and clear versions of while loops with the `for` construct:

  ```python
  >>> t = (2, 0, 9, 10, 11)
  >>> s = 0
  >>> k = 0
  >>> while k < len(t):
  ...     s += t[k]
  ...     k += 1
  >>> print(s)
  32
  >>> t = (2, 0, 9, 10, 11)
  >>> s = 0
  >>> for x in t:
  ...     s += x
  >>> print(s)
  32
  ```

- Iteration over numbers is really the same, conceptually:

  ```python
  >>> s = 0
  >>> k = 1
  >>> while k < 10:
  ...     s += k
  ...     k += 1
  >>> print(s)
  45
  >>> s = 0
  >>> for k in range(1, 10):
  ...     s += k
  ...     print(s)
  45
  ```