Chapter 1 : Lists and List Processing

Note that for resizable lists of integers, floats, and Strings, you can use the Processing classes IntList, FloatList, and StringList. An ArrayList is a resizable-array implementation of the Java List interface.

If the space reserved for the dynamic array is exceeded, it is reallocated and possibly copied, which is an expensive operation. Linked lists have several advantages over dynamic arrays. Insertion or deletion of an element at a specific point of a list, assuming that we have indexed a pointer to the node before the one to be removed, or before the insertion point already, is a constant-time operation otherwise without this reference it is O n , whereas insertion in a dynamic array at random locations will require moving half of the elements on average, and all the elements in the worst case. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", this causes fragmentation that impedes the performance of iteration. Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and will have to reallocateâ€”an expensive operation, one that may not even be possible if memory is fragmented, although the cost of reallocation can be averaged over insertions, and the cost of an insertion due to reallocation would still be amortized O 1. An array from which many elements are removed may also have to be resized in order to avoid wasting too much space. On the other hand, dynamic arrays as well as fixed-size array data structures allow constant-time random access , while linked lists allow only sequential access to elements. Singly linked lists, in fact, can be easily traversed in only one direction. Sequential access on arrays and dynamic arrays is also faster than on linked lists on many machines, because they have optimal locality of reference and thus make good use of data caching. Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values , because the storage overhead for the links may exceed by a factor of two or more the size of the data. In contrast, a dynamic array requires only the space for the data itself and a very small amount of control data. Some hybrid solutions try to combine the advantages of the two representations. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record. A good example that highlights the pros and cons of using dynamic arrays vs. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle n times. Once you reach the nth person, take them out of the circle and have the members close the circle. Then count around the circle the same n times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. However, the linked list will be poor at finding the next person to remove and will need to search through the list until it finds that person. A dynamic array, on the other hand, will be poor at deleting nodes or elements as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the nth person in the circle by directly referencing them by their position in the array. The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated and has been the subject of much research. A balanced tree has similar memory access patterns and space overhead to a linked list while permitting much more efficient indexing, taking O log n time instead of O n for a random access. However, insertion and deletion operations are more expensive due to the overhead of tree manipulations to maintain balance. Schemes exist for trees to automatically maintain themselves in a balanced state: AVL trees or red-black trees. Singly linked linear lists vs. A singly linked linear list is a recursive data structure, because it contains a pointer to a smaller object of the same type. For that reason, many operations on singly linked linear lists such as merging two lists, or enumerating the elements in reverse order often have very simple recursive algorithms, much simpler than any solution using iterative commands. While those recursive solutions can be adapted for doubly linked and circularly linked lists, the procedures generally need extra arguments and more complicated base cases. Linear singly linked lists also allow tail-sharing , the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new oneâ€”a simple example of a persistent data structure. Again, this is not true with the other variants: In particular, end-sentinel nodes can be shared among singly linked non-circular lists. The same end-sentinel node may be used for every such list. In Lisp , for example, every proper list ends with a link to a special node, denoted by nil or , whose CAR and CDR links point to itself. The advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes, at no extra cost. To do the same in a singly linked list, one must have the address of the pointer to that node, which is either the handle for the whole list in case of the first node or the link field in the previous node. Some algorithms require access in both directions. On the other hand, doubly linked lists do not allow tail-sharing and cannot be used as persistent data structures Circularly linked vs. In these applications, a pointer to any node serves as a handle to the whole list. With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list e. A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge. The simplest representation for an empty circular list when such a thing makes sense is a null pointer, indicating that the list has no nodes. Without this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty linear list is more natural and often creates fewer special cases. Using sentinel nodes[edit] Sentinel node may simplify certain list operations, by ensuring that the next or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. Another example is the merging two sorted lists: However, sentinel nodes use up extra space especially in applications that use many short lists , and they may complicate other operations such as the creation of a new empty list. However, if the circular list is used merely to simulate a linear list, one may avoid some of this complexity by adding a single sentinel node to every list, between the last and the first data nodes. With this convention, an empty list consists of the sentinel node alone, pointing to itself via the next-node link. The list handle should then be a pointer to the last data node, before the sentinel, if the list is not empty; or to the sentinel itself, if the list is empty. The same trick can be used to simplify the handling of a doubly linked linear list, by turning it into a circular doubly linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use null to refer to an end-of-list marker or sentinel , which may be implemented in a number of ways. Linearly linked lists[edit] Singly linked lists[edit] Our node data structure will have two fields. We also keep a variable firstNode which always points to the first node in the list, or is null for an empty list. The diagram shows how it works. Inserting a node before an existing one cannot be done directly; instead, one must keep track of the previous node and insert a node after it. This requires updating firstNode. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element. Inserting to a list before a specific node requires traversing the list, which would have a worst case running time of O n. Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the List structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly linked lists are each of length n , list appending has asymptotic time complexity of O.

Chapter 2 : Linked List Processing

Lists are designed to have some of the features of ArrayLists, but to maintain the simplicity and efficiency of working with arrays. Functions like sort() and shuffle() always act on the list itself. To get a sorted copy, use calendrierdelascience.com().sort().

Integers and floats are numeric types, which means they hold numbers. We can use the numeric operators we saw last chapter with them to form numeric expressions. The Python interpreter can then evaluate these expressions to produce numeric values, making Python a very powerful calculator. Strings, lists, and tuples are all sequence types, so called because they behave like a sequence - an ordered collection of objects. Squence types are qualitatively different from numeric types because they are compound data types - meaning they are made up of smaller pieces. There is also the empty string, containing no characters at all. In the case of lists or tuples, they are made up of elements, which are values of any Python datatype, including other lists and tuples. Lists are enclosed in square brackets [and] and tuples in parentheses and. A list containing no elements is called an empty list, and a tuple with no elements is an empty tuple. The third is a tuple containing four integers, followed by a tuple containing four strings. The last is a list containing three tuples, each of which contains a pair of strings. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful. Note It is possible to drop the parentheses when specifiying a tuple, and only use a comma seperated list of values: The expression inside brackets is called the index, and must be an integer value. The index indicates which element to select, hence its name. Think of the index as the numbers on a ruler measuring how many elements you have moved into the sequence from the beginning. Both rulers and indices start at 0. Now that you have seen the len function, you might be tempted to try something like this: It causes the runtime error IndexError: The reason is that len seq returns the number of elements in the list, 16, but there is no element at index position 16 in seq. Since we started counting at zero, the sixteen indices are numbered 0 to To get the last element, we have to subtract 1 from the length: The expression seq[-1] yields the last element, seq[-2] yields the second to last, and so on. The most common pattern is to start at the beginning, select each element in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. For now just note that the colon: Sometimes it is helpful to have both the value and the index of each element. The enumerate function gives us this: Like with indexing, we use square brackets [ ] as the slice operator, but instead of one integer value inside we have two, seperated by a colon: This behavior is counter-intuitive; it makes more sense if you imagine the indices pointing between the characters, as in the following diagram: If you omit the first index before the colon , the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. When you slice a sequence, the resulting subsequence always has the same type as the sequence from which it was derived. This is not generally true with indexing, except in the case of strings. It evaluates to True if one string is a substring of another: Also note that computer programmers like to think about these edge cases quite carefully! Each string in the above examples is followed by a dot operator, a method name, and a parameter list, which may be empty. Invoking the method causes an action to take place using the value on which the method is invoked. Since each of the characters in the string represents a digit, the isdigit method returns the boolean value True. The strip removes leading and trailing whitespace. You can find out more about each of these methods by printing out their docstrings. To find out what the replace method does, for example, we do this: If the optional argument count is given, only the first count occurrences are replaced. Using this information, we can try using the replace method to varify that we know how it works. Optional arguments start and end are interpreted as in slice notation. Raises ValueError if the value is not present. We will explore these functions in the exercises. Using the bracket operator on the left side of an assignment, we can update one of the elements: An assignment to an element of a list is called item assignment. Item assignment does not work for strings: Python provides an alternative that is more readable. You can use a slice as an index for del: Since lists are mutable, these methods modify the list on which they are invoked, rather than returning a new list. There are two possible states: In the second case, they refer to the same object.

Because the same list has two different names, a and b, we say that it is aliased. Since lists are mutable, changes made with one alias affect the other: In general, it is safer to avoid aliasing when you are working with mutable objects. This process is sometimes called cloning, to avoid the ambiguity of the word copy. The easiest way to clone a list is to use the slice operator: In this case the slice happens to consist of the whole list. Now we are free to make changes to b without worrying about a: In this list, the element with index 3 is a nested list: To extract an element from the nested list, we can proceed in two steps: The list command takes a sequence type as an argument and creates a list out of its elements. When applied to a string, you get a list of characters. The default delimiter is whitespace, which includes spaces, tabs, and newlines. The join method does approximately the oposite of the split method. It takes a list of strings as an argument and returns a string of all the list elements joined together. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b: Python provides a form of tuple assignment that solves this problem neatly: Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile. Naturally, the number of variables on the left and the number of values on the right have to be the same: He created the mathematics we call Boolean algebra , which is the basis of all modern computer arithmetic. There are only two boolean values, True and False. The semantics meaning of these operators is similar to their meaning in English. This can be quite useful in preventing runtime errors. Imagine you want check if the fifth number in a tuple of integers named numbers is even. The following expression will work: Traceback most recent call last: Since it uses short-circuit evaluation, it does not, and the runtime error is avoided. For the numeric and sequence types we have seen thus far, truthiness is defined as follows: Combining this notion of truthiness with an understanding of short-circuit evaluation makes it possible to understand what Python is doing in the following expressions: Boolean values result when a boolean expression is evaluated by the Python interepreter. They have type bool. Elements have a value and an index. Assignments to elements or slices of immutable types cause a runtime error. Methods are invoked on the object using the dot operator. All mutable types are compound types. Lists and dictionaries are mutable data types; strings and tuples are not. Objects have both data values and behaviors methods. More generally, a subsequence of any sequence type in Python can be created using the slice operator sequence[start: The third and optional argument to the range function is called the step size. If not specified, it defaults to 1. Tuples can be used wherever an immutable type is required, such as a key in a dictionary see next chapter.

Chapter 3 : Mailing List Management Services | Business | Consumer

We offer high quality, personalized laser printing, ink jet addressing onto plain or gloss surfaces, folding, inserting, tabbing and more. We also provide a full range of list services, including data entry, mail presort, CASS processing (for automation postage rates), NCOA and LACS address correction, mail list merging and duplicate removal.

Hy Connection to artificial intelligence[edit] Since inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP [14] systems. In the s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue. Moreover, each given dialect may have several implementationsâ€”for instance, there are more than a dozen implementations of Common Lisp. Differences between dialects may be quite visibleâ€”for instance, Common Lisp uses the keyword defun to name a function, but Scheme uses define. Historically significant dialects[edit] 4. So named because it contained several improvements on the original "LISP 1" interpreter, but was not a major restructuring as the planned LISP 2 would be. It was rendered obsolete by Maclisp and InterLisp. It ran on the PDP and Multics systems. For quite some time, Maclisp and InterLisp were strong competitors. ZetaLisp had a big influence on Common Lisp. LeLisp is a French Lisp dialect. EuLisp â€” attempt to develop a new efficient and cleaned-up Lisp. The Language as a base document and to work through a public consensus process to find solutions to shared issues of portability of programs and compatibility of Common Lisp implementations. ACL2 is both a programming language which can model computer systems, and a tool to help proving properties of those models. Clojure , a recent dialect of Lisp which compiles to the Java virtual machine and has a particular focus on concurrency. It was written using Allegro Common Lisp and used in the development of the entire Jak and Daxter series of games. Most new activity is focused around implementations of Common Lisp , Scheme , Emacs Lisp , Clojure , and Racket , and includes development of new portable libraries and applications. Raymond to pursue a language others considered antiquated. New Lisp programmers often describe the language as an eye-opening experience and claim to be substantially more productive than in other languages. The open source community has created new supporting infrastructure: CLiki is a wiki that collects Common Lisp related information, the Common Lisp directory lists resources, lisp is a popular IRC channel and allows the sharing and commenting of code snippets with support by lisppaste , an IRC bot written in Lisp , Planet Lisp collects the contents of various Lisp-related blogs, on LispForum users discuss Lisp topics, Lispjobs is a service for announcing job offers and there is a weekly news service, Weekly Lisp News. Quicklisp is a library manager for Common Lisp. The Scheme community actively maintains over twenty implementations. Several significant new implementations Chicken, Gambit, Gauche, Ikarus, Larceny, Ypsilon have been developed in the last few years. The Scheme Requests for Implementation process has created a lot of quasi standard libraries and extensions for Scheme. User communities of individual Scheme implementations continue to grow. A new language standardization process was started in and led to the R6RS Scheme standard in Academic use of Scheme for teaching computer science seems to have declined somewhat. Some universities, such as MIT, are no longer using Scheme in their computer science introductory courses. The parser for Julia is implemented in Femtolisp, a dialect of Scheme Julia is inspired by Scheme, and is often considered a Lisp. Major dialects[edit] Common Lisp and Scheme represent two major streams of Lisp development. These languages embody significantly different design choices. Common Lisp is a successor to Maclisp. Common Lisp is a general-purpose programming language and thus has a large language standard including many built-in data types, functions, macros and other language elements, and an object system Common Lisp Object System. Common Lisp also borrowed certain features from Scheme such as lexical scoping and lexical closures. It was designed to have exceptionally clear and simple semantics and few different ways to form expressions. Designed about a decade earlier than Common Lisp, Scheme is a more minimalist design. It has a much smaller set of standard features but with certain implementation features such as tail-call optimization and full continuations not specified in Common Lisp. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme. Scheme continues to evolve with a series of standards Revisedn Report on the Algorithmic Language Scheme and a series of Scheme Requests for Implementation. It is designed to be a pragmatic general-purpose language. Clojure draws considerable influences from Haskell and places a very strong emphasis on immutability. The potential small size of a useful Scheme interpreter makes it particularly popular for embedded scripting. Thus, Lisp functions can be manipulated, altered or even created within a Lisp program without lower-level manipulations. This is generally considered one of the main advantages of the language with regard to its expressive power, and makes the language suitable for syntactic macros and metacircular evaluation. A conditional using an if-then-else syntax was invented by McCarthy in a Fortran context. For Lisp, McCarthy used the more general cond-structure. Lisp deeply influenced Alan Kay , the leader of the research team that developed Smalltalk at Xerox PARC ; and in turn Lisp was influenced by Smalltalk, with later dialects adopting object-oriented programming features inheritance classes, encapsulating instances, message passing, etc. The Flavors object system introduced the concept of multiple inheritance and the mixin. The Common Lisp Object System provides multiple inheritance, multimethods with multiple dispatch , and first-class generic functions , yielding a flexible and powerful form of dynamic dispatch. It has served as the template for many subsequent Lisp including Scheme object systems, which are often implemented via a metaobject protocol , a reflective metacircular design in which the object system is defined in terms of itself: Lisp was only the second language after Smalltalk and is still one of the very few languages to possess such a metaobject system. Many years later, Alan Kay suggested that as a result of the confluence of these features, only Smalltalk and Lisp could be regarded as properly conceived object-oriented programming systems. Progress in modern sophisticated garbage collection algorithms such as generational garbage collection was stimulated by its use in Lisp. Dijkstra in his Turing Award lecture said, "With a few very basic principles at its foundation, it [LISP] has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of in a sense our most sophisticated computer applications. I think that description a great compliment because it transmits the full flavour of liberation: Because of its suitability to complex and dynamic applications, Lisp is enjoying some resurgence of popular interest in the s. Symbolic expressions S-expressions [edit] Lisp is an expression oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements" ;[dubious â€" discuss] all code and data are written as expressions. When an expression is evaluated, it produces a value in Common Lisp, possibly multiple values , which can then be embedded into other expressions. Each value can be any data type. Symbolic expressions S-expressions , sexps , which mirror the internal representation of code and data; and Meta expressions M-expressions , which express functions of S-expressions. M-expressions never found favor, and almost all Lisps today use S-expressions to manipulate both code and data. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. The reliance on expressions gives the language great flexibility. Because Lisp functions are written as lists, they can be processed exactly like data. This allows easy writing of programs which manipulate other programs metaprogramming. Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit. Lists[edit] A Lisp list is written with its elements separated by whitespace , and surrounded by parentheses. For example, 1 2 foo is a list whose elements are the three atoms 1, 2, and foo. These values are implicitly typed: The empty list is also represented as the special atom nil. This is the only entity in Lisp which is both an atom and a list. Expressions are written as lists, using prefix notation. The first element in the list is the name of a function, the name of a macro, a lambda expression or the name of a "special operator" see below. The remainder of the list are the arguments. For example, the function list returns its arguments as a list, so the expression list 1 2 quote foo evaluates to the list 1 2 foo. The "quote" before the foo in the preceding example is a "special operator" which returns its argument without evaluating it. Any unquoted expressions are recursively evaluated before the enclosing expression is evaluated. For example, list 1 2 list 3 4 evaluates to the list 1 2 3 4. Note that the third argument is a list; lists can be nested. Arithmetic operators are treated similarly. Lisp has no notion of operators as implemented in Algol-derived languages. Arithmetic operators in Lisp are variadic functions or n-ary , able to take any number of arguments. For example, the special operator if takes three arguments. If the first argument is non-nil, it evaluates to the second argument; otherwise, it evaluates to the third argument.

Thus, the expression if nil list 1 2 "foo" list 3 4 "bar" evaluates to 3 4 "bar". Of course, this would be more useful if a non-trivial expression had been substituted in place of nil. Lisp also provides logical operators and, or and not. The and and or operators do short circuit evaluation and will return their first nil and non-nil argument respectively. Lambda expressions and function definition[edit] Another special operator, lambda, is used to bind variables to values which are then evaluated within an expression. This operator is also used to create functions: Lambda expressions are treated no differently from named functions; they are invoked the same way. Named functions are created by storing a lambda expression in a symbol using the defun macro. It is conceptually similar to the expression: A list was a finite ordered sequence of elements, where each element is either an atom or a list, and an atom was a number or a symbol. A symbol was essentially a unique named item, written as an alphanumeric string in source code , and used either as a variable name or as a data item in symbolic processing.

Chapter 4 : ABAP LEAVE TO LIST-PROCESSING Statement syntax and functionality in SAP

They Called It LISP for a Reason: List Processing. Lists play an important role in Lisp--for reasons both historical and practical. Historically, lists were Lisp's original composite data type, though it has been decades since they were its only such data type.

List Processing Lists play an important role in Lisp--for reasons both historical and practical. These days, a Common Lisp programmer is as likely to use a vector, a hash table, or a user-defined class or structure as to use a list. Do not try and bend the list. There is no list. There is no list? Those simpler objects are pairs of values called cons cells, after the function CONS used to create them. CONS takes two arguments and returns a new cons cell containing the two values. Unless the second value is NIL or another cons cell, a cons is printed as the two values in parentheses separated by a dot, a so-called dotted pair. At the dawn of time, these names were mnemonic, at least to the folks implementing the first Lisp on an IBM But even then they were just lifted from the assembly mnemonics used to implement the operations. Lists are built by linking together cons cells in a chain. The elements of the list are held in the CARs of the cons cells while the links to subsequent cons cells are held in the CDRs. However, few languages outside the Lisp family provide such extensive support for this humble data type. The CAR of the cons cell is the first item of the list, and the CDR is a reference to another list, that is, another cons cell or NIL, containing the remaining elements. The Lisp printer understands this convention and prints such chains of cons cells as parenthesized lists rather than as dotted pairs. Box-and-arrow diagrams represent cons cells as a pair of boxes like this: The values stored in a particular cons cell are either drawn in the appropriate box or represented by an arrow from the box to a representation of the referenced value. And a single list can hold objects of different types. Because lists can have other lists as elements, you can also use them to represent trees of arbitrary depth and complexity. As such, they make excellent representations for any heterogeneous, hierarchical data. Another obvious example of tree-structured data is Lisp code itself. Common Lisp provides quite a large library of functions for manipulating lists. However, they will be easier to understand in the context of a few ideas borrowed from functional programming. Functional Programming and Lists The essence of functional programming is that programs are built entirely of functions with no side effects that compute their results based solely on the values of their arguments. The advantage of the functional style is that it makes programs easier to understand. Eliminating side effects eliminates almost all possibilities for action at a distance. And since the result of a function is determined only by the values of its arguments, its behavior is easier to understand and test. Functions that deal with numbers are naturally functional since numbers are immutable. However, lists can be treated as a functional data type if you consider their value to be determined by the elements they contain. Thus, any list of the form 1 2 3 4 is functionally equivalent to any other list containing those four values, regardless of what cons cells are actually used to represent the list. And any function that takes a list as an argument and returns a value based solely on the contents of the list can likewise be considered functional. The reason most list functions are written functionally is it allows them to return results that share cons cells with their arguments. To take a concrete example, the function APPEND takes any number of list arguments and returns a new list containing the elements of all its arguments. One obvious way to achieve that goal is to create a completely new list consisting of four new cons cells. Instead, APPEND actually makes only two new cons cells to hold the values 1 and 2, linking them together and pointing the CDR of the second cons cell at the head of the last argument, the list 3 4. It then returns the cons cell containing the 1. None of the original cons cells has been modified, and the result is indeed the list 1 2 3 4. The resulting structure looks like this: In general, APPEND must copy all but its last argument, but it can always return a result that shares structure with the last argument. Others are simply allowed to return shared structure at the discretion of the implementation. However, using the same term to describe all state-modifying operations leads to a certain amount of confusion since there are two very different kinds of destructive operations, for-side-effect operations and recycling operations. However, if you mix nonfunctional, for-side-effect operations with functions that return structure-sharing results, then you need to be careful not to inadvertently modify the shared structure. For instance, consider these three definitions: On the other hand, the other kind of destructive operations, recycling operations, are intended to be used in functional code. They use side effects only as an optimization. In particular, they reuse certain cons cells from their arguments when building their result. However, unlike functions such as APPEND that reuse cons cells by including them, unmodified, in the list they return, recycling functions reuse cons cells as raw material, modifying the CAR and CDR as necessary to build the desired result. But suppose you write something like this: No new cons cells need to be allocated, and no garbage is created. In general, the recycling functions have names that are the same as their non-destructive counterparts except with a leading N. However, the waters are further muddied by a handful of recycling functions with specified side effects that can be relied upon. It then returns the first list, which is now the head of the spliced-together result. To make matters worse, shared structure and recycling functions tend to work at cross-purposes. Nondestructive list functions return lists that share structure under the assumption that cons cells are never modified, but recycling functions work by violating that assumption. In practice, recycling functions tend to be used in a few idiomatic ways. By far the most common recycling idiom is to build up a list to be returned from a function by "consing" onto the front of a list, usually by PUSHing elements onto a list stored in a local variable and then returning the result of NREVERSEing it. Other uses are possible but require keeping careful track of which functions return shared structure and which do not. Following that rule will, of course, rule out using any destructive functions, recycling or otherwise. In either case you need to be sure to save the result of the sorting function because the original argument is likely to be in tatters. More generally, the function NTH takes two arguments, an index and a list, and returns the nth zero-based element of the list. In other words, these are really functions on trees rather than lists: And even the most die-hard old-school Lisp hackers tend to avoid the longer combinations. With an integer, argument returns the last n cons cells. With an integer argument, excludes the last n cells. The initial elements of the list are NIL or the value specified with the: No reliable side effects. ATOM Predicate to test whether an object is not a cons cell. Functionally equivalent to NOT but stylistically preferable when testing for an empty list as opposed to boolean false. Mapping Another important aspect of the functional style is the use of higher-order functions, functions that take other functions as arguments or return functions as values. You saw several examples of higher-order functions, such as MAP, in the previous chapter. Although MAP can be used with both lists and vectors that is, with any kind of sequence , Common Lisp also provides six mapping functions specifically for lists. The differences between the six functions have to do with how they build up their result and whether they apply the function to the elements of the list or to the cons cells of the list structure. Instead, its first argument is the function to apply, and subsequent arguments are the lists whose elements will provide the arguments to the function. Otherwise, it behaves like MAP: The results of each function call are collected into a new list. Thus, each function invocation can provide any number of elements to be included in the result. This actually corresponds well with how many Common Lisp implementations work--although all objects are conceptually stored by reference, certain simple immutable objects can be stored directly in a cons cell. Another difference is that ELT will signal an error if you try to access an element at an index greater than or equal to the length of the list, but NTH will return NIL. For example, you could take apart the following expression:

Chapter 5 : Strings, lists, and tuples â€” Beginning Python Programming for Aspiring Web Developers

List is the cartesian product of the lists in LList, that is, the list of lists formed with one element of each list in LList, in the same order. The following properties should hold at call time: (term_typing:nonvar/1) LList is currently a term which is not a free variable.

List Processing Introduction This tutorial is a large one, because it covers a big topic: We cover several of the available modes of the zl object, which provides a central clearinghouse of list processing functions. We also see how we can perform mathematical expressions against lists with the vexpr object, and how to use the prob object to create a table of probabilities for music creation. Within Max, the list is one of the most powerful data structures available. You can define any combination of values in a list, then have them sent as messages, get processed by objects or get treated as small tables. Much of this is made possible by the zl object, which give you the ability to query, reorder and access any of the elements in a list. Adding vexpr into the list processing mix allows you to process list elements mathematically without having to break them into their individual components. The use of the prob object is integral to many generative programs. The prob object allows you to easily create a weighted probability table, with only a bang required to generate the expected data. In our tutorial, we will see how the prob object is used to seed a basic sequencing application. Running the sequencing patch Open the tutorial. Take a look at our tutorial. This is the quintessential Max patch â€” the step sequencer. When you open the patch, the first two multislider objects labeled A and B are seeded with values, while the third labeled C is empty. These objects contain sequences for a MIDI playback system fed by the two noteout objects to the right of them. Double-click the noteout objects to select an available synthesizer. Turn on the metro at the top-left by clicking the toggle , and the sequence will begin playing, as driven by the A and B multislider contents. If you are using a General MIDI-compatible synthesizer, you should hear the top two multislider sequences as drums, and the bottom C sequence as a melody played on a piano. This is because, according to General MIDI rules, MIDI channel 10 the argument to the upper noteout is the drum channel, while the other fifteen channels play melody instruments. Most synthesizers enable you to configure this behavior, but the built-in synthesizers on both Macintosh and Windows machines behave in this way. You can rotate the sequence stored in multislider A, reverse the contents of multislider B, and perform several different functions against the melody contained in multislider C. Click on the button objects that activate the editing routines, and see that the changes are immediately applied against the multislider contents. The basic function of the step sequencer should be familiar: The two drum channels provide a simplified way of selecting drums â€” the top two multislider objects are limited to only three values each. In this way, the full range of the multislider has significance, and the output value is easy to work with. Set the entire sequence in the A and B multislider objects to 0 drag the mouse across them towards the bottom. The drums in the sequence will stop. Try adding different patterns to get a sense of how the numbers in the object map to the different sounds you get. Working with prob probabilities The initial setup of the drum sequences is done using the generate new patterns section of the patch, found below the multislider objects. You will see that the A multislider will get scrambled, but that the contents tend to emphasize the 1 value which corresponds to the closed hi-hat. This occurs because of the probability table set up and enforced by the prob object. The setup of the probability table is done through the large message triggered when the patcher opened by the loadbang. The way that our probability table works is based on transition: So, for example, 0 1 2 means that the probability applied to a transition from 0 to 1 is 2. What does 2 mean? As a result, we consider it the weight of that transition. The message box used for the hi-hat probability tables sets up all of the possible transitions within a single three value set. Since you can only be at one current location at a time, the probabilities are dependent on the current value state. In this example, the total weighting values for all cases where the current step is 0 add up to 8. This gives us the denominator for the calculations we made. If you look at the cases when the current step is 2 an open hi-hat , you will see that the transition to 0 and 1 both are weighted at 4, while the transition to 2 a repeat of the open hat is given a weight of 0. The prob object is driven by an uzi object, set to output 32 bang messages and therefore 32 values. However, these are presented as individual values, and the multislider is expecting a value list for setup. How do we combine all these messages efficiently? Working with zl A key object when working with lists is the zl object: In the case of the sequence generators for the drums, the zl object is used in group mode, with an argument of This means that the object will collect 32 values, group them into a single list, and then output the list from its left outlet. This is a quick and very efficient way to pack a stream of data into a list of predefined, and turns an otherwise onerous task into a job for a single object. The tutorial patch shows many uses of the zl object â€" particularly in the editing functions. All of these editing routines start with the zl object in reg mode. In this mode, the zl object will accept a list into its right inlet, and store it until it receives a bang in the left inlet. The zl reg object is, in essence, the equivalent of the int , float or value objects, but is designed for list storage. When you click on the button for each editing routine, it outputs the list from the zl reg into another list processing object. Most zl modes take a second argument or allow a number in the right inlet to change a parameter - for example, the zl iter object breaks a list into sub-lists of a size set by the argument in our case, 4 that can also be changed by sending a number into the right inlet. If we change the number box connected to the zl iter object to 32 the entire length of the sequence and then click the button , the melody stored in multislider C will become sorted from low values to high values. Perhaps the best way to see all of the zl operating modes is to look at the zl help file. The help file is broken into three segments, since there are so many modes â€" but you will be able to see all of the ways that zl can be used to bend, fold and mutilate Max lists. Working with vexpr When working with lists, there are times when you want to perform mathematical expressions against all of the elements in a list. There is a solution: As you can tell by the name, the vexpr object is very similar to the expr object we learned about earlier, with the exception that it is made specifically to process lists. Therefore, when the zl reg object outputs a value list sent originally from multislider C , each of the entries will be incremented or decremented, and will be output as a value list. The right-hand side of the routine will generate a value list of random numbers varying from -1 to 1 the result of the uzi , random 3 object and the â€" 1 objects sent into a zl group. This is input into the right inlet of vexpr. The left inlet receives the value list from multislider C being held by a zl reg. This means that each of the list elements will be altered by a different random element, giving a small randomization to the multislider contents. NOte that this randomization can be done multiple times to gradually morph the melody into a new pattern altogether. Since lists are an integral data structure in the Max environment, these objects will become an often-used part of your programming toolkit.

Chapter 6 : c# - Processing Lists and calendrierdelascience.com() - Stack Overflow

View Notes - Linked Lists and Linked List Processing from ICS 46 at University of California, Irvine. Linked List - linear linked list Linked List Basics Simple templated class LN (List Nodes) in.

Chapter 7 : Processing a list of lists in Python - Stack Overflow

In this chapter we will be extending our knowledge of C to include the important topics of lists and list processing. In an earlier chapter we looked at arrays and how they could be used to store data, and we noted some of the restrictions inherent in these data structures: for example, the need to declare an array large enough to hold all the data likely to be needed in any one run of the program.

Chapter 8 : Max Data Tutorial 5: List Processing

First, we will study various linked list processing methods in isolation (or in some unspecified collection class, with instance variables). By the end of this lecture we will examine the List interface, and how it can be implemented directly via arrays and linked lists: these classes contains a large number of methods that perform interesting manipulations of arrays/lists.

Chapter 9 : Linked list - Wikipedia

What you think are list you generate with Take and OrderBy are IEnumerables. IEnumerable s are not containers like List. They are nothing but a promise to yield data when enumerated.