A pattern can be

a wild card: `_`	match anything, bind nothing
an *identifier*	match anything, bind the identifier.
ident `@ pattern`	*ident* binds to the whole match
A nulluary constructor (*e.g.*, `[]`, `Nothing`, `True`)	need to match, nothing to bind
Explcit tuples of patterns (*e.g.*, `()`, `(x,y)`)	matches a tuple value using the corresponding pattern for each piece
An explcit list of patterns (*e.g.*, `[\,x]`), including string constants	Matches a list of the same length
a constructor followed by patterns (*e.g.*, `Just x`)	matches a value made by that constructor
`~` followed by a pattern,	`~` delays pattern matching
`!` followed by a pattern.	`!` forces the matched value
with pragma {`-# LANGUAGE ViewPatterns #-`} *`fn -> pattern`*	apply *`fn`* to what would have matched there, then match *`pattern`* against that.
with pragma {`-# LANGUAGE PatternSynonyms #-`} a unser defined pattern	search the web for `Haskell pattern synonyms`. See §6.7.4 of the Glasgow Haskell Ccmpiler Guide.

Where? Patterns only occur in specific locations:

- To the left of `=` ○ in top-level assignment, ○ in a `where` block, and ○ in a `let-in` expression.
- To the left of `->`.
  - In an anonymous function, `\ pat -> ...`
  - in `case ... of { pat -> expr ;...}` expressions
- To the left of `<-`. · in `do` blocks, · list compreneions, and · in guard expressions.
- To the right of `->` in view patterns (see below)
- On the right hand side of pattern definition (see below)

Warning

- Haskell re-uses syntax, so, for instance, `[a]` may be an expression, a pattern, or a type, depending on where it occurs.
- Constructors (from `data` and `newtype` declarations) explicitly provide both patterns (for taking values apart) and expressions (for bluding values).

Purpose:

1. Patterns can be used to bind identifiers (variables) to values. In "`f x = x+2`" the first `x` is a pattern. In evaluating `f(5)`, the pattern `x` is bound to `5`, and then `x+2` is evaluated.
2. Patterns can choose between different computation scenarios.

   ```
   length [] = 0
   length (_:xs) = 1 + length xs
   ```

   The pattern `[]` matches on the empty list, whereas `[_:xs]` matches only non-empty lists (and binds `xs` to the list tail).

View Patterns

- Use the pragma `{-# LANGUAGE ViewPatterns #-}` at the top of a file where you wish to use these.
- Allows patterns to have one more syntax: `(fn) -> ptn1`, in any place where a pattern is allowed. The function expression `fn` is applied to whatever the pattern originally would have matched, and then `ptn1` is matched against it. For instance

  ```
  bitSum 0 = 0
  bitSum ((`divMod` 2) -> (k,b)) = b + bitSum k
  ```

Pattern Synonyms

- require `{-# LANGUAGE PatternSynonyms #-}`
- allow creating brand-new patterns.
- Example:

  ```
  pattern FirstTwo x1 x2 <- (x1:x2:_)
  ```

  creates a pattern that matches any list of length two or longer, for instance one can write

  ```
  \ xs - case xs of
        FirstTwo a b -> show a ++ show b ++ " ..."
        [q]          -> "Singleton" ++ show q
        []           -> "Empty list"
  ```

- the details of pattern synonyms are tricky but the idea is very powerful.
