`funarray` Package

This package provides some convinient functional functions for typst to use on arrays. Let us import the package and define

```
#import "@preview/funarray:0.3.0"
```

chunks

The chunks function translates the array to an array of array. It groups the elements to chunks of a given size and collects them in an bigger array.

```
let a = (1, "not prime", 2, "prime", 3, "prime", 4, "not prime", 5, "prime")
funarray.chunks(a, 2)
```

return type: array

```
(
  (1, "not prime"),
  (2, "prime"),
  (3, "prime"),
  (4, "not prime"),
  (5, "prime"),
)
```

unzip

The unzip function is the inverse of the zip method, it transforms an array of pairs to a pair of vectors.

```
let a = (
  (1, "not prime"),
  (2, "prime"),
  (3, "prime"),
  (4, "not prime"),
  (5, "prime"),
)
funarray.unzip(a)
```

return type: array

```
(
  (1, 2, 3, 4, 5),
  (
    "not prime",
    "prime",
    "prime",
    "not prime",
    "prime",
  ),
)
```

cycle

The cycle function concatenates the array to itself until it reaches a given size.

```
let a = range(5)
funarray.cycle(a, 8)
```

return type: array

```
(0, 1, 2, 3, 4, 0, 1, 2)
```

Note that there is also the functionality to concatenate with + and * in typst.

windows and circular-windows

This function provides a running window

```
let a = range(5)
funarray.windows(a, 3)
```

return type: array

```
((0, 1, 2), (1, 2, 3), (2, 3, 4))
```

whereas the circular version wraps over.

```
let a = range(5)
funarray.circular-windows(a, 3)
```

return type: array

```
(
  (0, 1, 2),
  (1, 2, 3),
  (2, 3, 4),
  (3, 4, 2),
  (4, 2, 3),
  (2, 3, 4),
)
```

partition and partition-map

The partition function seperates the array in two according to a predicate function. The result is an array with all elements, where the predicate returned true followed by an array with all elements, where the predicate returned false.

```
let a = (
  (1, "not prime"),
  (2, "prime"),
  (3, "prime"),
  (4, "not prime"),
  (5, "prime"),
)
let (primes, nonprimes) = funarray.partition(a, x => x.at(1) == "prime")
primes
```

return type: array

```
((2, "prime"), (3, "prime"), (5, "prime"))
```

There is also a partition-map function, which after partition also applies a second function on both collections.

```
let a = (
  (1, "not prime"),
  (2, "prime"),
  (3, "prime"),
  (4, "not prime"),
  (5, "prime"),
)
let (primes, nonprimes) = funarray.partition-map(
  a,
  x => x.at(1) == "prime",
  x => x.at(0)
)
primes
```

return type: array

```
(2, 3, 5)
```

group-by

This functions groups according to a predicate into maximally sized chunks, where all elements have the same predicate value.

```
let f = (0,0,1,1,1,0,0,1)
funarray.group-by(f, x => x == 0)
```

return type: array

```
((0, 0), (1, 1, 1), (0, 0), (1,))
```

flatten

Typst has a `flatten` method for arrays, however that method acts recursively. For instance

```
(((1,2,3), (2,3)), ((1,2,3), (1,2))).flatten()
```

return type: array

```
(1, 2, 3, 2, 3, 1, 2, 3, 1, 2)
```

Normally, one would only have flattened one level. To do this, we can use the typst array concatenation method `+`, or by folding, the `sum` method for arrays:

```
(((1,2,3), (2,3)), ((1,2,3), (1,2))).sum()
```

return type: array

```
((1, 2, 3), (2, 3), (1, 2, 3), (1, 2))
```

To handle further depth, one can use flatten again, so that in our example:

```
(
  ((1,2,3), (2,3)), ((1,2,3), (1,2))
).sum().sum() == (
  ((1,2,3), (2,3)), ((1,2,3), (1,2))
).flatten()
```

return type: boolean

```
true
```

take-while and skip-while

These functions do exactly as they say.

```
#let f = (0,0.5,0.2,0.8,2,1,0.1,0,-2,1)
#funarray.take-while(f, x => x < 1)

#funarray.skip-while(f, x => x < 1)
```

```
(0, 0.5, 0.2, 0.8)
(2, 1, 0.1, 0, -2, 1)
```

accumulate and scan

These functions are similar to `fold`, but produce again arrays and do not reduce into one final value.

Compare

```
#let a = (1,) * 10
#a.fold(0, (acc, x) => acc + x)

#funarray.accumulate(a, 0, (acc, x) => acc + x)

#funarray.scan(a, (0, 0), (acc, x) => {
  let s = acc.sum() + x
```

```
  ((acc.at(1), s), s)
})
```

```
10
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
(1, 2, 4, 7, 12, 20, 33, 54, 88, 143)
```

`accumulate` gives us also intermediate results compared to `fold`. Scan is even more flexible, by simulating mutable state (here `acc`), we can look e.g. back 2 states.

The signatures are as follows

Function	Signature	Accumulators $\mathcal{A}$
`fold`	$B^n \times A \times \mathcal{A} \to A$	$f : A \times B \to A$
`accumulate`	$B^n \times A \times \mathcal{A} \to A^n$	$f : A \times B \to A$
`scan`	$B^n \times A \times \mathcal{A} \to C^n$	$f : A \times B \to A \times C$

where B^n is the inputed array and A our initial value/state.

unfold and iterated

These functions can be used to construct array. `iterated` applies a function multiple times, so that

```
funarray.iterated(1, x => 2*x, 10)
```
return type: `array`

```
(2, 4, 8, 16, 32, 64, 128, 256, 512, 1024)
```

We provided a initial value, a function and the length of the resulting array. Similarly, but more powerful is `unfold`

```
funarray.unfold((1, 1), x => {
  let next = x.sum()
  ((x.at(1), next), x.at(0))
}, 10)
```
return type: `array`

```
(1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
```

Here we created the first fibonacci numbers by passing the previous two values as state.

The signatures are as follows

Function	Signature	Argument Functions $\mathcal{F}$
`iterated`	$A \times \mathcal{F} \times \{n\} \to A^n,\ n \in \mathbb{N}$	$f : A \to A$
`unfold`	$A \times \mathcal{F} \times \{n\} \to B^n,\ n \in \mathbb{N}$	$f : A \to A \times B$

where A is the initial value/state.

Unsafe functions

The core functions are defined in `funarray-unsafe.typ`. However, assertions (error checking) are not there and it is generally not being advised to use these directly. Still, if being cautious, one can use the imported `funarray-unsafe` module in `funarray(.typ)`. All function names are the same.

To do this from the package, do as follows:

```
#funarray.funarray-unsafe.chunks(range(10), 3)
```

```
((0, 1, 2), (3, 4, 5), (6, 7, 8), (9,))
```