On this page

Getting Started

Welcome to the Taichi Language documentation!

Installation

To get started with the Taichi Language, simply install it with :`pip`

```
python3 -m pip install taichi
```

Copy

> **NOTE**
>
> Currently, Taichi only supports Python 3.6/3.7/3.8/3.9 (64-bit).

Feedback

There are a few of extra requirements depend on which operating system you are using:

Arch Linux **Windows**

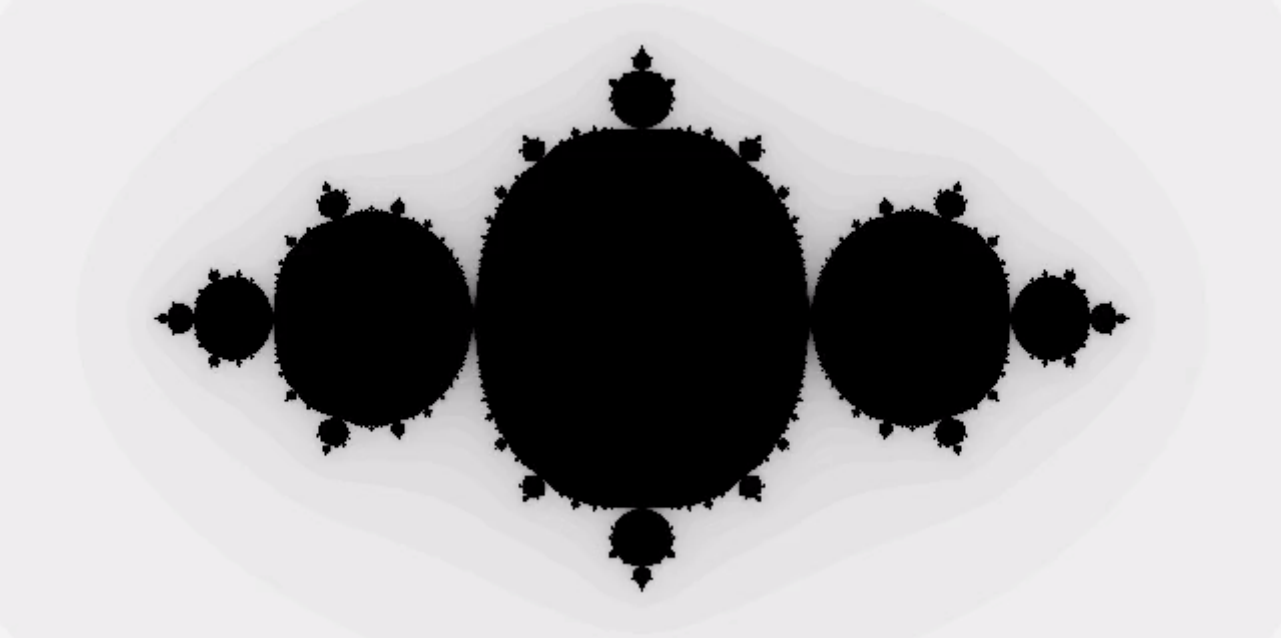
On Arch Linux, you need to install package from the Arch User Repository: `ncurses5-compat-libs` `yaourt -S ncurses5-compat-libs`

Please refer to the Installation Troubleshooting section if you run into any issues when installing Taichi.

Hello, world!

We introduce the Taichi programming language through a very basic *fractal* example.

Running the Taichi code below using either or *(you can find more information about the Taichi CLI in the Command line utilities section)* will give you an animation of Julia set:`python3 fractal.py` `ti example fractal`



fractal.py

```python
import taichi as ti

ti.init(arch=ti.gpu)

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallelized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
```

```
26      paint(i * 0.03)
27      gui.set_image(pixels)
28      gui.show()
```

Let's dive into this simple Taichi program.

import taichi as ti

Taichi is a domain-specific language (DSL) embedded in Python.

To make Taichi as easy to use as a Python package, we have done heavy engineering with this goal in mind - letting every Python programmer write Taichi programs with minimal learning effort.

You can even use your favorite Python package management system, Python IDEs and other Python packages in conjunction with Taichi.

```
1  # Run on GPU, automatically detect backend
2  ti.init(arch=ti.gpu)
3
4  # Run on GPU, with the NVIDIA CUDA backend
5  ti.init(arch=ti.cuda)
6  # Run on GPU, with the OpenGL backend
7  ti.init(arch=ti.opengl)
8  # Run on GPU, with the Apple Metal backend, if you are on macOS
9  ti.init(arch=ti.metal)
10
11 # Run on CPU (default)
12 ti.init(arch=ti.cpu)
```


INFO

Supported backends on different platforms:

platform	CPU	CUDA	OpenGL	Metal	C source
Windows	OK	OK	OK	N/A	N/A

platform	CPU	CUDA	OpenGL	Metal	C source
Linux	OK	OK	OK	N/A	OK
macOS	OK	N/A	N/A	OK	N/A

(OK: supported; N/A: not available)

With , Taichi will first try to run with CUDA. If CUDA is not supported on your machine, Taichi will fall back on Metal or OpenGL. If no GPU backend (CUDA, Metal, or OpenGL) is supported, Taichi will fall back on CPUs.`arch=ti.gpu`

NOTE

When used with the CUDA backend on Windows or ARM devices (e.g., NVIDIA Jetson), Taichi allocates 1 GB GPU memory for field storage by default.

You can override this behavior by initializing with to allocate GB GPU memory, or to allocate of the total GPU memory.`ti.init(arch=ti.cuda, device_memory_GB=3.4)` `3.4` `ti.init(arch=ti.cuda, device_memory_fraction=0.3)` `30%`

On other platforms, Taichi will make use of its on-demand memory allocator to allocate memory adaptively.

Feedback

Fields

Taichi is a **data**-oriented programming language where dense or spatially-sparse fields are the first-class citizens.

In the code above, allocates a 2D dense field named of size and element data type .`pixels = ti.field(dtype=float, shape=(n * 2, n))` `pixels` `(640, 320)` `float`

Functions and kernels

Computation resides in Taichi **kernels** and Taichi **functions**.

Taichi **kernels** are defined with the decorator . They can be called from Python to perform computation. Kernel arguments must be type-hinted (if any).`@ti.kernel`

Taichi **functions** are defined with the decorator . They can **only** be called by Taichi kernels or other Taichi functions.`@ti.func`

See syntax for more details about Taichi kernels and functions.

The language used in Taichi kernels and functions looks exactly like Python, yet the Taichi frontend compiler converts it into a language that is **compiled, statically-typed, lexically-scoped, parallel and differentiable**.

> **INFO**
>
> **Taichi-scopes v.s. Python-scopes**:
>
> Everything decorated with and is in Taichi-scope and hence will be compiled by the Taichi compiler.`@ti.kernel` `@ti.func`
>
> Everything else is in Python-scope. They are simply Python native code.

> **CAUTION**
>
> Taichi kernels must be called from the Python-scope. Taichi functions must be called from the Taichi-scope.

> **TIP**
>
> For those who come from the world of CUDA, corresponds to while corresponds to .`ti.func` `__device__` `ti.kernel` `__global__`

> **NOTE**
>
> Nested kernels are **not supported**.
>
> Nested functions are **supported**.
>
> Recursive functions are **not supported for now**.

Parallel for-loops

For loops at the outermost scope in a Taichi kernel is **automatically parallelized**. For loops can have two forms, i.e. *range-for loops* and *struct-for loops*.

Range-for loops are no different from Python for loops, except that they will be parallelized when used at the outermost scope. Range-for loops can be nested.

```
@ti.kernel
def fill():
    for i in range(10): # Parallelized
        x[i] += i

        s = 0
        for j in range(5): # Serialized in each parallel thread
            s += j

        y[i] = s

@ti.kernel
def fill_3d():
    # Parallelized for all 3 <= i < 8, 1 <= j < 6, 0 <= k < 9
    for i, j, k in ti.ndrange((3, 8), (1, 6), 9):
        x[i, j, k] = i + j + k
```


NOTE

It is the loop **at the outermost scope** that gets parallelized, not the outermost loop.

```
@ti.kernel
def foo():
    for i in range(10): # Parallelized :-)
        ...

@ti.kernel
def bar(k: ti.i32):
    if k > 42:
        for i in range(10): # Serial :-(
            ...
```

Struct-for loops are particularly useful when iterating over (sparse) field elements. In the above, loops over all the pixel coordinates, i.e., `fractal.py` `for i, j in pixels` `(0, 0), (0, 1), (0, 2), ... , (0, 319), (1, 0), ..., (639, 319)`

> **NOTE**
>
> Struct-for is the key to sparse computation in Taichi, as it will only loop over active elements in a sparse field. In dense fields, all elements are active.

> **CAUTION**
>
> Struct-for loops must live at the outer-most scope of kernels.
>
> It is the loop **at the outermost scope** that gets parallelized, not the outermost loop.

```
@ti.kernel
def foo():
    for i in x:
        ...

@ti.kernel
def bar(k: ti.i32):
    # The outermost scope is a `if` statement
    if k > 42:
        for i in x: # Not allowed. Struct-fors must live in the outermost scope.
            ...
```

> **CAUTION**
>
> `break` **is not supported in parallel loops**:

```
@ti.kernel
def foo():
  for i in x:
      ...
```

```
        break # Error!
    for i in range(10):
        ...
        break # Error!

@ti.kernel
def foo():
    for i in x:
        for j in range(10):
            ...
            break # OK!
```

GUI system

Taichi provides a cpu-based GUI system for users to render their results on the screen.


```
gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
  paint(i * 0.03)
  gui.set_image(pixels)
  gui.show()
```

Interacting with other Python packages

Python-scope data access

Everything outside Taichi-scopes (and) is simply Python code. In Python-scopes, you can access Taichi field elements using plain indexing syntax. For example, to access a single pixel of the rendered image in Python-scope, you can simply use:`ti.func` `ti.kernel`

```
import taichi as ti
pixels = ti.field(ti.f32, (1024, 512))
```

```
pixels[42, 11] = 0.7  # store data into pixels
print(pixels[42, 11]) # prints 0.7
```

Sharing data with other packages

Taichi provides helper functions such as and to transfer data between Taichi fields and NumPy arrays, so that you can also use your favorite Python packages (e.g., , ,) together with Taichi as below: `from_numpy` `to_numpy` `numpy` `pytorch` `matplotlib`

```
import taichi as ti
pixels = ti.field(ti.f32, (1024, 512))

import numpy as np
arr = np.random.rand(1024, 512)
pixels.from_numpy(arr)   # load numpy data into taichi fields

import matplotlib.pyplot as plt
arr = pixels.to_numpy()  # store taichi data into numpy arrays
plt.imshow(arr)
plt.show()

import matplotlib.cm as cm
cmap = cm.get_cmap('magma')
gui = ti.GUI('Color map')
while gui.running:
    render_pixels()
    arr = pixels.to_numpy()
    gui.set_image(cmap(arr))
    gui.show()
```


See Interacting with external arrays for more details.

What's next?

Now we have gone through core features of the Taichi programming language using the fractal example, feel free to dive into the language concepts in the next section, or jump

to the advanced topics, such as the Metaprogramming or Differentiable programming. Remember that you can use the search bar at the top right corner to search for topics or keywords at any time!

If you are interested in joining the Taichi community, we strongly recommend you take some time to familiarize yourself with our contribution guide.

We hope you enjoy your adventure with Taichi!

Edit this page

*Last updated on **1/27/2022** by **Taichi Gardener***