INDIAN INSTITUTE OF TECHNOLOGY MADRAS

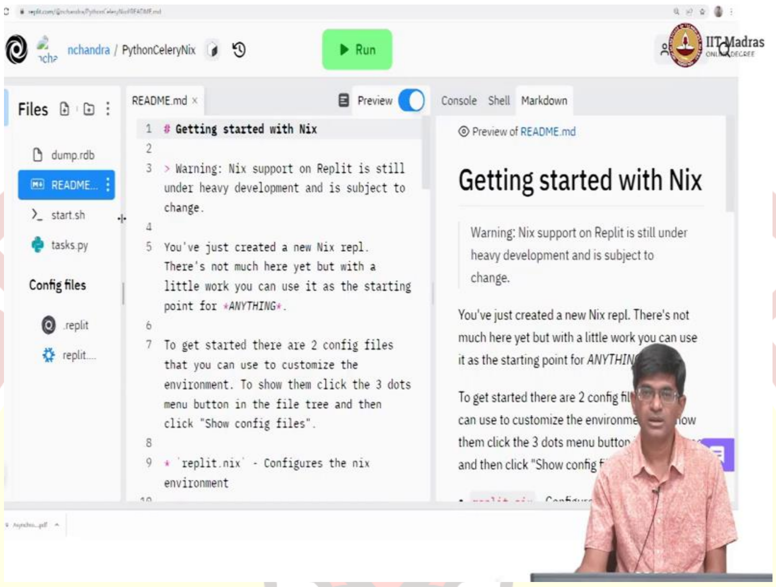
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Modern Application Development - II
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
How to use Celery in Replit?

(Refer Slide Time: 0:14)



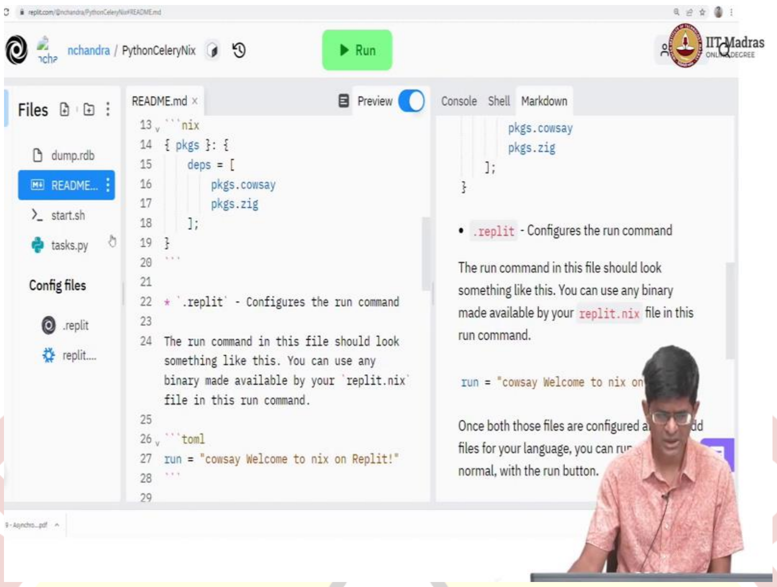
```
# Getting started with Nix

> Warning: Nix support on Replit is still
under heavy development and is subject to
change.

You've just created a new Nix repl.
There's not much here yet but with a
little work you can use it as the starting
point for *ANYTHING*.

To get started there are 2 config files
that you can use to customize the
environment. To show them click the 3 dots
menu button in the file tree and then
click "Show config files".

* `replit.nix` - Configures the nix
environment
```

Hello, everyone, welcome to modern application development part 2. So, in order to give a demonstration of how celery can be used on the replit platform, I am just going to walk through a simple example, where we are using a package manager called Nix in order to set up and run a Redis server.

Now, to be frank, the usage of Nix and how it is to be used and how you can actually manage a lot of packages with it in the user space and so on, is probably overkill, it is not really something that you explicitly need to know in order to develop applications. So, this is most definitely what could be considered a slightly advanced topic. But, especially for people who are interested in application development moving forward, knowing about such systems helps you in setting up larger scale environments that can be done and in a reproducible manner.

So, some of these things are worth sort of at least being aware of, even if you do not use them a whole lot. In this particular case, this replica that I am walking through, has a few different things.

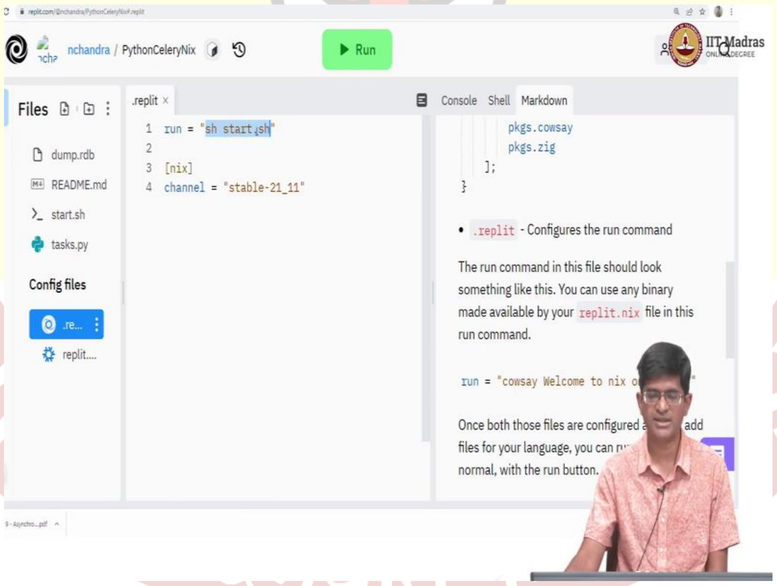
(Refer Slide Time: 1:27)



~~~
```nix
{ pkgs }: {
    deps = [
        pkgs.cowsay
        pkgs.zig
    ];
}
~~~

* `.replit` - Configures the run command

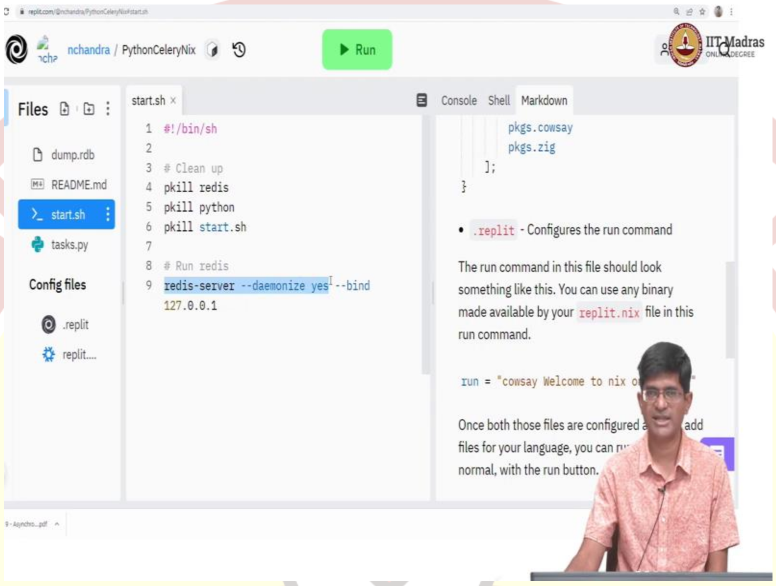
The run command in this file should look
something like this. You can use any
binary made available by your `replit.nix`
file in this run command.

```toml
run = "cowsay Welcome to nix on Replit!"
```
```

```
        pkgs.cowsay
        pkgs.zig
    ];
}
```

• .replit - Configures the run command

The run command in this file should look something like this. You can use any binary made available by your replit.nix file in this run command.

```
run = "cowsay Welcome to nix on
```

Once both those files are configured a   dd files for your language, you can run normal, with the run button.

Of course, it has a readme, which gives a little bit of information about Nix, how to get started with Nix and so on. But that is more sort of general information from replit, unrelated to the question of how to actually run the program that we are interested in.

(Refer Slide Time: 1:44)


nchandra / PythonCeleryNix
Run
IIT Madras
Files
dump.rdb
README.md
start.sh
tasks.py
Config files
.re...
replit....
.replit
Console
Shell
Markdown

```
run = "sh start.sh"

[nix]
channel = "stable-21_11"
```

```
        pkgs.cowsay
        pkgs.zig
    ];
}
```

• .replit - Configures the run command

The run command in this file should look something like this. You can use any binary made available by your replit.nix file in this run command.

```
run = "cowsay Welcome to nix o
```

Once both those files are configured a   add files for your language, you can ru normal, with the run button.

So, for us, what we have is, there are a few important files. There is a new something called a config file, which is named just dot replit, which has a little bit of information over here. It basically says, run equals sh start dot sh. So, all that it is saying is when I run this report, this is the command to run, so you will normally wonder when I click on the Run, command Run button, in any Replit, how does it know what to do?
```

By default, if you have set up your replit, as let us say, a Python application, it runs main dot pi. If you have set it up as HTML plus CSS application, it will open index dot HTML. So, those are some kind of defaults that replit uses. But this is a new kind of application a replit that we are creating, which is a Nix based application. So, we have to explicitly tell it, what we want it to do. And what we say is run this shell script.

(Refer Slide Time: 2:45)



```
#!/bin/sh

# Clean up
pkill redis
pkill python
pkill start.sh

# Run redis
redis-server --daemonize yes --bind
127.0.0.1
```

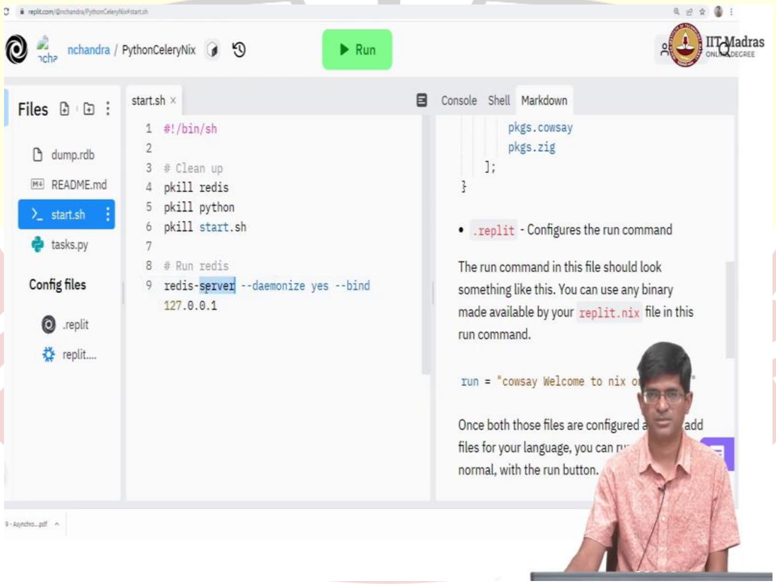
```
        pkgs.cowsay
        pkgs.zig
    ];
}
```

- `.replit` - Configures the run command

The run command in this file should look something like this. You can use any binary made available by your `replit.nix` file in this run command.

```
run = "cowsay Welcome to nix o
```

Once both those files are configured a add files for your language, you can r normal, with the run button.



```
#!/bin/sh

# Clean up
pkill redis
pkill python
pkill start.sh

# Run redis
redis-server --daemonize yes --bind
127.0.0.1
```

```
        pkgs.cowsay
        pkgs.zig
    ];
}
```

- `.replit` - Configures the run command

The run command in this file should look something like this. You can use any binary made available by your `replit.nix` file in this run command.

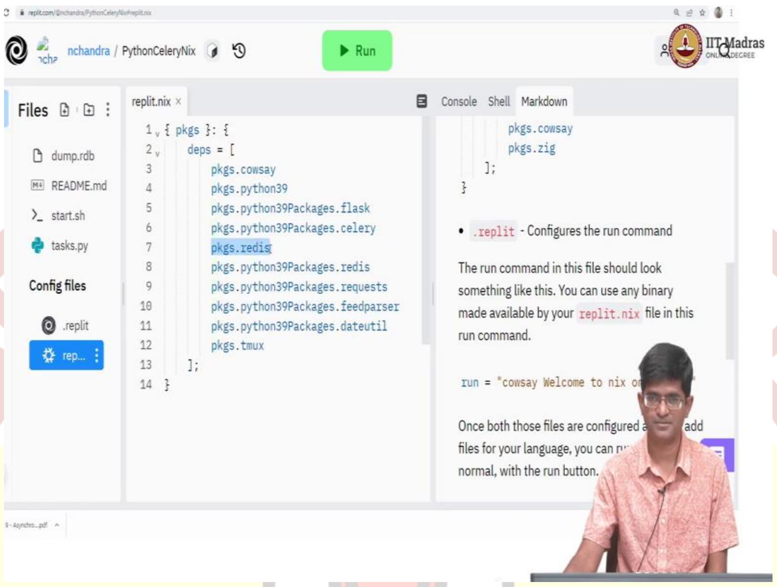
```
run = "cowsay Welcome to nix o
```

Once both those files are configured a add files for your language, you can r normal, with the run button.

What is inside that shell script is that it will basically kill off any pre-existing versions of either Redis anything to do with Redis, or anything to do with Python, or this shell script itself. And then finally, it will just run this command Redis server in the background. So, this daemonize yes, is basically saying you run it in the background.

Now, where did Redis server come from? Because this is not something that's normally installed on any Linux system, which means that I must have had a way by which I can explicitly run it and install it.

(Refer Slide Time: 3:22)



```
{ pkgs }: {
    deps = [
        pkgs.cowsay
        pkgs.python39
        pkgs.python39Packages.flask
        pkgs.python39Packages.celery
        pkgs.redis
        pkgs.python39Packages.redis
        pkgs.python39Packages.requests
        pkgs.python39Packages.feedparser
        pkgs.python39Packages.dateutil
        pkgs.tmux
    ];
}
```

```
            pkgs.cowsay
            pkgs.zig
        ];
}
```

- `.replit` - Configures the run command

The run command in this file should look something like this. You can use any binary made available by your `replit.nix` file in this run command.

```
run = "cowsay Welcome to nix on
```

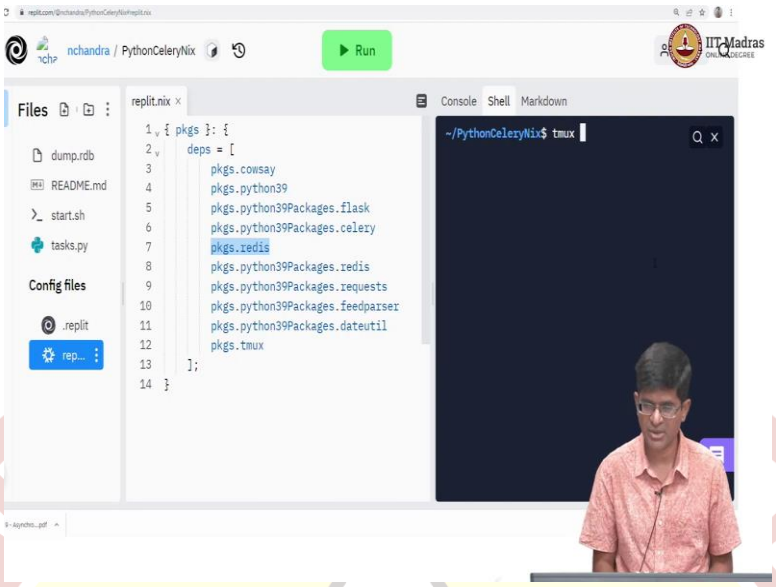
Once both those files are configured a add files for your language, you can ru normal, with the run button.

In this case, there is also one file called replit dot nix, which is sort of the crucial thing that we have here. You do not need to worry about the syntax, you do not need to know about what exactly is happening here. The important thing is that just by having this replit dot nix file, a few things get automatically installed into this replit.

One of them is Python 3.9. In addition to that, the flask module for Python 3.9, the salary module for Python 3.9, and a few other packages of Python packages, including ones to communicate with redis, one for generating HTTP requests, one for parsing, RSS feeds, actually, this requests and feed parser in date, until are not really used in this code.

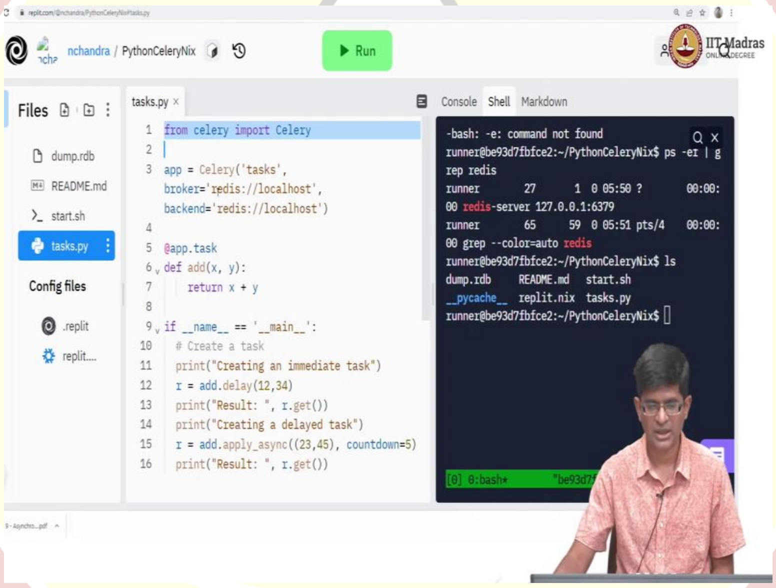
But I just put it in there for it was copied from the example that replit was using, and they were present over there. So, I have just left them in case we want to build on this code. I am also adding on another package called tmux, which is a terminal multiplexer. It will allow me to run multiple things at the same time using a single shell. And most importantly, this package called redis, so what is redis that is the one that actually has the Redis server and the Redis CLI. So, what this is doing in other words, is that anytime I open up this system, and if I click on Run.

(Refer Slide Time: 4:47)



```
{ pkgs }: {
    deps = [
        pkgs.cowsay
        pkgs.python39
        pkgs.python39Packages.flask
        pkgs.python39Packages.celery
        pkgs.redis
        pkgs.python39Packages.redis
        pkgs.python39Packages.requests
        pkgs.python39Packages.feedparser
        pkgs.python39Packages.dateutil
        pkgs.tmux
    ];
}
```

```
~/PythonCeleryNix$ tmux
```



```
from celery import Celery

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
  print("Result: ", r.get())
```

```
-bash: -e: command not found
runner@be93d7fbfce2:~/PythonCeleryNix$ ps -er | g
rep redis
runner       27       1  0 05:50 ?        00:00:
00 redis-server 127.0.0.1:6379
runner       65      59  0 05:51 pts/4    00:00:
00 grep --color=auto redis
runner@be93d7fbfce2:~/PythonCeleryNix$ ls
dump.rdb     README.md   start.sh
__pycache__  replit.nix  tasks.py
runner@be93d7fbfce2:~/PythonCeleryNix$ 
[0] 0:bash*          "be93d7
```

It runs this start dot sh and it has installed all the packages corresponding to this Nix file that I have over here, which means at this point in time, I now have Redis server available to me in the shell, so I can basically go open up the shell. And this is where the terminal multiplexer becomes useful, because what I do is I basically just run tmux. And it creates, you can see this green line at the bottom over here, which basically is some kind of, it is like a window frame. It is telling me that there are multiple different windows available to me.

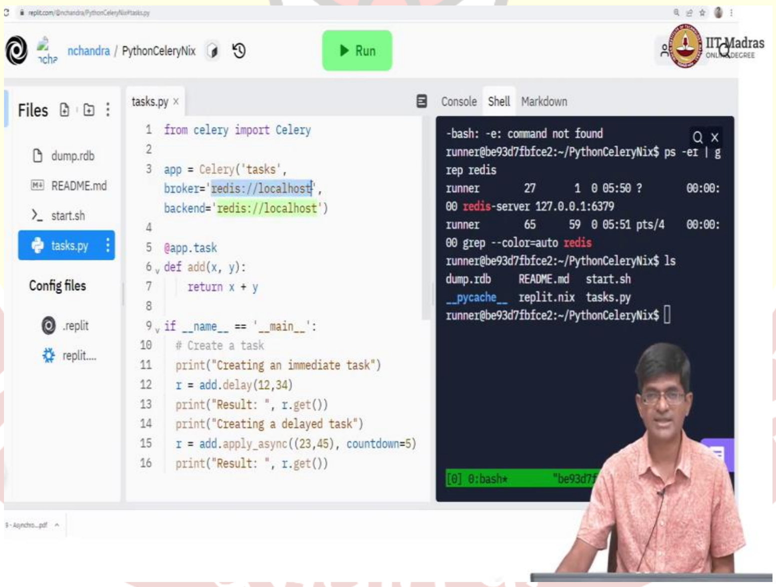
Once again, explaining how tmux works is out of scope of what I can cover over here. But it is worth understanding, especially if you find yourself needing to remote log into a system and actually run multiple commands older. So, what can I do? First things first, I can check whether I already have a Redis server running.

And sure enough, I find that there is actually a Redis server already running in the background. Why did that happen? Because I clicked on the run button, and it ran that program start dot sh. So, that is good to know. What I have over here is now there is this file tasks dot py.

And in tasks dot py, what I find is I have imported salary. And I have created a new flask. Well, it is not a flask application, this is just a pure Python application. I am not really trying to use flask at this point. So, this demonstration that I am giving here is just with Python salary, and nothing to do with flask, you could very easily add flask to it. But that is not what I am going to do in this demo.

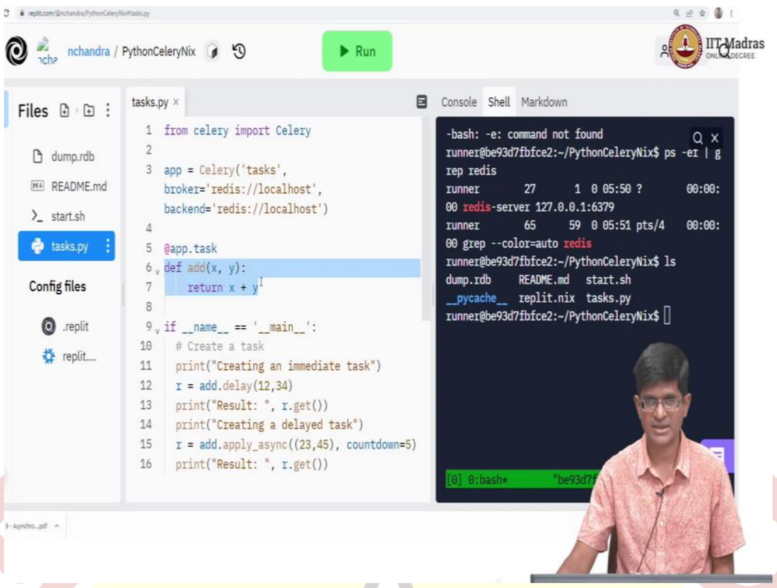
What does it do? It just basically creates a salary instance. It gives it the name tasks, which is the same as this module tasks dot py that I am currently running with. And there are two pieces of information the message broker, which is used for sending messages between any Python code that you write, and the celery workers that are going to receive those messages and work with them.

(Refer Slide Time: 7:01)



```python
from celery import Celery

app = Celery('tasks',
    broker='redis://localhost',
    backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
    # Create a task
    print("Creating an immediate task")
    r = add.delay(12,34)
    print("Result: ", r.get())
    print("Creating a delayed task")
    r = add.apply_async((23,45), countdown=5)
    print("Result: ", r.get())
```

```
-bash: -e: command not found
runner@be93d7fbfce2:~/PythonCeleryNix$ ps -er | g
rep redis
runner       27       1  0 05:50 ?        00:00:
00 redis-server 127.0.0.1:6379
runner       65      59  0 05:51 pts/4    00:00:
00 grep --color=auto redis
runner@be93d7fbfce2:~/PythonCeleryNix$ ls
dump.rdb     README.md   start.sh
__pycache__  replit.nix  tasks.py
runner@be93d7fbfce2:~/PythonCeleryNix$
[0] 0:bash*         "be93d7f
```



```python
from celery import Celery

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
  print("Result: ", r.get())
```

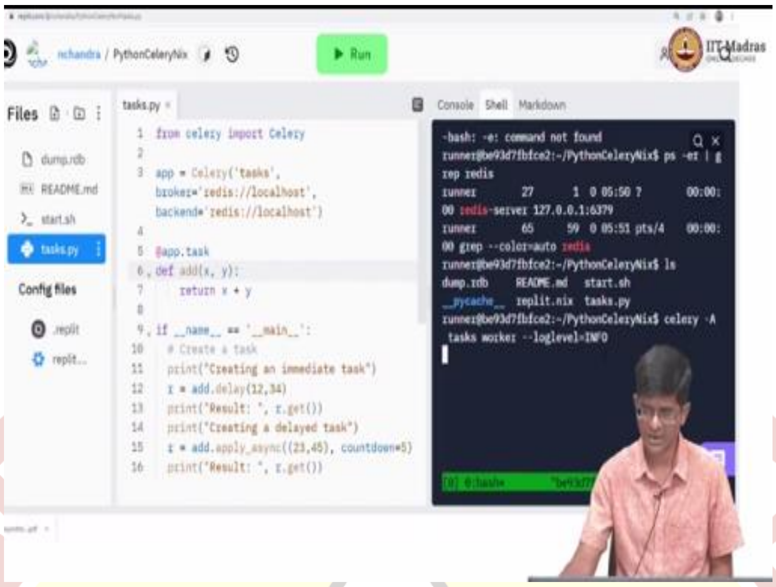
```
-bash: -e: command not found
runner@be93d7fbfce2:~/PythonCeleryNix$ ps -er | g
rep redis
runner       27       1  0 05:50 ?        00:00:
00 redis-server 127.0.0.1:6379
runner       65      59  0 05:51 pts/4    00:00:
00 grep --color=auto redis
runner@be93d7fbfce2:~/PythonCeleryNix$ ls
dump.rdb    README.md   start.sh
__pycache__  replit.nix  tasks.py
runner@be93d7fbfce2:~/PythonCeleryNix$
```

I say I point that to the Redis instance that I have, which I have already started. Same way there is also something called a backend, what is the backend for? As in when a task gets completed, the result of that task needs to come back. That also needs to be sent back through some kind of database or some system, I am using the same redis for that as well, I could have done this with RabbitMQ, in which case, I would have needed to add RabbitMQ as one of the dependencies and actually start it up ahead of time.

But since Redis, can handle both the broker and the backend, I am just using Redis for simplicity. Now, how does celery actually work? You need to define certain tasks write using this decorator at app dot task. What is the task over here, something completely trivial just add two numbers.

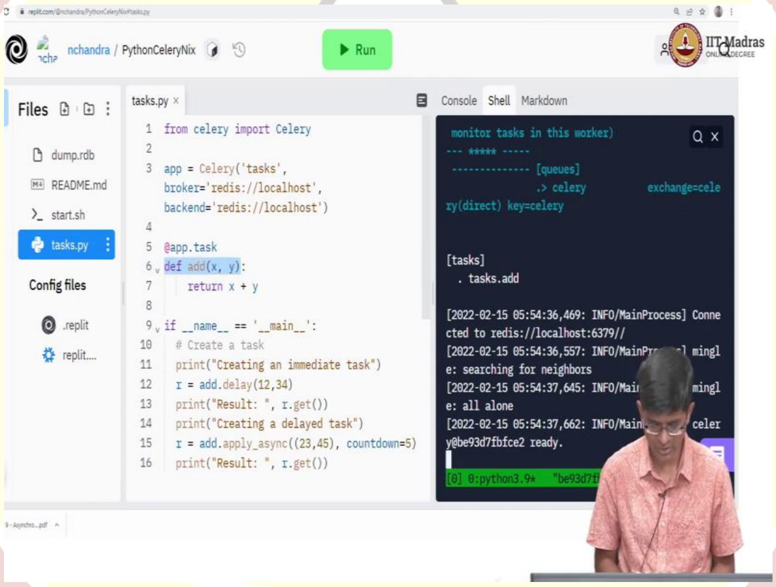
So, in practice, of course, this would never be a task, because like I said earlier, the time required for actually executing this task is much less than the time required for pushing things onto the queue, pulling them off the queue, executing them pushing the result back, and so on.

(Refer Slide Time: 8:12)



```python
from celery import Celery

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
    # Create a task
    print("Creating an immediate task")
    r = add.delay(12,34)
    print("Result: ", r.get())
    print("Creating a delayed task")
    r = add.apply_async((23,45), countdown=5)
    print("Result: ", r.get())
```

```
-bash: -e: command not found
runner@be93d7fbfce2:~/PythonCeleryNix$ ps -er | g
rep redis
runner        27       1  0 05:50 ?        00:00:
00 redis-server 127.0.0.1:6379
runner        65      59  0 05:51 pts/4    00:00:
00 grep --color=auto redis
runner@be93d7fbfce2:~/PythonCeleryNix$ ls
dump.rdb     README.md   start.sh
__pycache__  replit.nix  tasks.py
runner@be93d7fbfce2:~/PythonCeleryNix$ celery -A
 tasks worker --loglevel=INFO
```



```python
from celery import Celery

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
    # Create a task
    print("Creating an immediate task")
    r = add.delay(12,34)
    print("Result: ", r.get())
    print("Creating a delayed task")
    r = add.apply_async((23,45), countdown=5)
    print("Result: ", r.get())
```

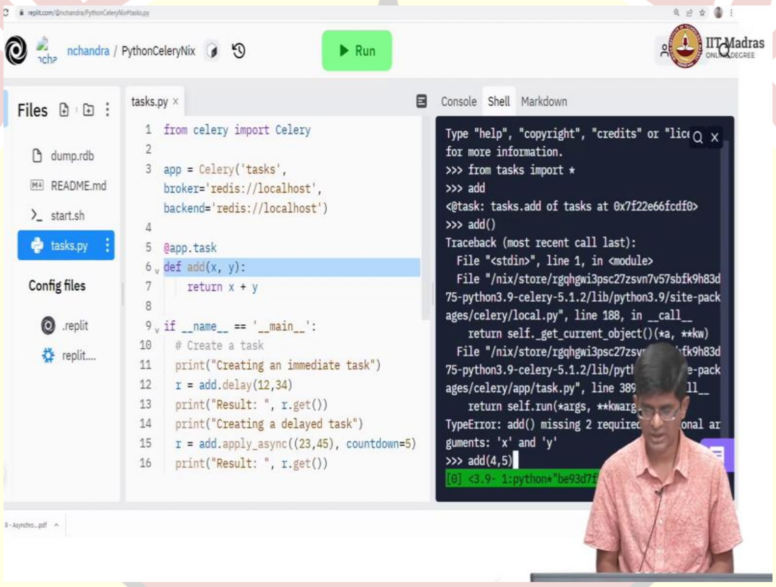
```
 monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=cele
ry(direct) key=celery


[tasks]
  . tasks.add

[2022-02-15 05:54:36,469: INFO/MainProcess] Conne
cted to redis://localhost:6379//
[2022-02-15 05:54:36,557: INFO/MainPr[illegible]] mingl
e: searching for neighbors
[2022-02-15 05:54:37,645: INFO/Main[illegible] mingl
e: all alone
[2022-02-15 05:54:37,662: INFO/Main[illegible] celer
y@be93d7fbfce2 ready.
[0] 0:python3.9*  "be93d7f[illegible]
```

So, now that I have this, what I can do is that I can actually run a command, celery and tell it to essentially, create work and rather instantiate workers from this module tasks. And I am going to give a log level just so that I get additional debugging information, or additional information printed on the screen. And what I have is that salary is supposed to start up. And yes, you can see that it has started up and some amount of information because after all, I had put it in info logging.

So, it says that it has now got tasks dot add, which came up from the, so in other words, what it did was it took this tasks dot py, because I had given minus capital A task dot py, that is just the standard syntax of celery. And what it is doing is it has started workers, a worker or multiple workers, that part is dependent on some configuration parameters. I am just going with defaults at this point. So the workers are now running over there. How I make use of this? So, what are all the things that are now running in the background?

There is a Redis server that is running in the background. And there is a celery worker that has just been started by me over here. And the Redis server is going to act both as the message broker and the message backend to take messages from my Python code into Celery and to get the results brought back from celery into my Python code, how I make use of this? Let me go ahead and create out of this over here, I can go back to the tmux. And create a new shell over there.

(Refer Slide Time: 10:05)

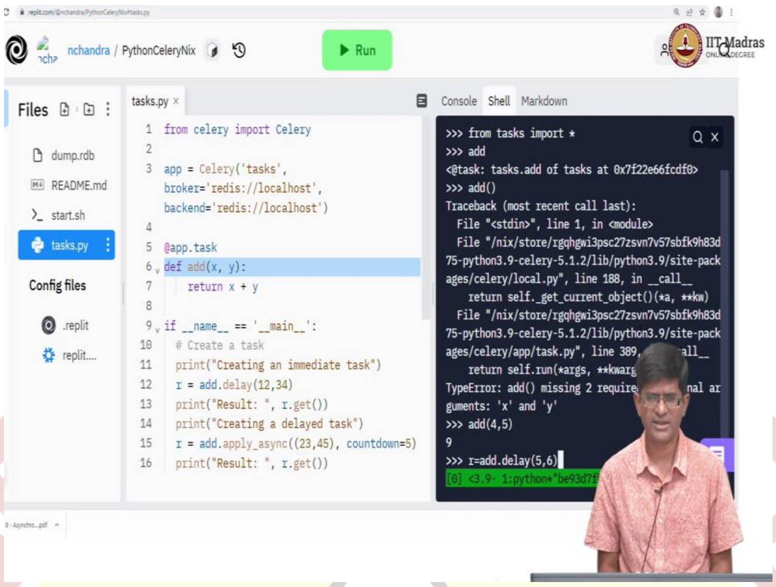


```python
from celery import Celery

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
  print("Result: ", r.get())
```

```
Type "help", "copyright", "credits" or "lic
for more information.
>>> from tasks import *
>>> add
<@task: tasks.add of tasks at 0x7f22e66fcdf0>
>>> add()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/nix/store/rgqhgwi3psc27zsvn7v57sbfk9h83d
75-python3.9-celery-5.1.2/lib/python3.9/site-pack
ages/celery/local.py", line 188, in __call__
    return self._get_current_object()(*a, **kw)
  File "/nix/store/rgqhgwi3psc27zsv      fk9h83d
75-python3.9-celery-5.1.2/lib/pyth        e-pack
ages/celery/app/task.py", line 389        ll__
    return self.run(*args, **kwarg
TypeError: add() missing 2 required       onal ar
guments: 'x' and 'y'
>>> add(4,5)
[0] <3.9- 1:python*"be93d7f
```

And I am just going to start an interactive Python terminal, where what I will do is I will import everything from tasks dot py. What is there to import? It will just import this one function add, which means that I should now be able to call add. And what is add? It basically says add is a task, can I directly call add? It says it is missing arguments? What happens if I give add 4, 5?

(Refer Slide Time: 10:39)

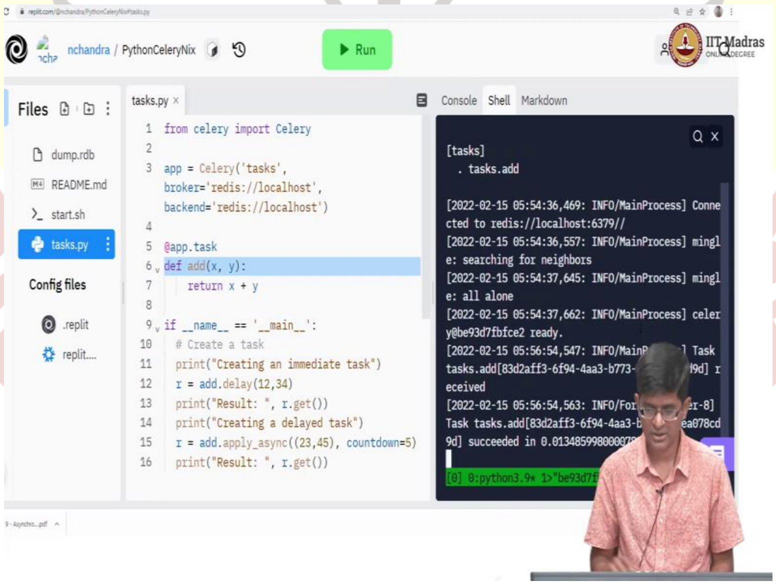


```python
from celery import Celery

app = Celery('tasks',
    broker='redis://localhost',
    backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
  print("Result: ", r.get())
```

```
>>> from tasks import *
>>> add
<@task: tasks.add of tasks at 0x7f22e66fcdf0>
>>> add()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/nix/store/rgqhgwi3psc27zsvn7v57sbfk9h83d
75-python3.9-celery-5.1.2/lib/python3.9/site-pack
ages/celery/local.py", line 188, in __call__
    return self._get_current_object()(*a, **kw)
  File "/nix/store/rgqhgwi3psc27zsvn7v57sbfk9h83d
75-python3.9-celery-5.1.2/lib/python3.9/site-pack
ages/celery/app/task.py", line 389,        all__
    return self.run(*args, **kwarg
TypeError: add() missing 2 require          nal ar
guments: 'x' and 'y'
>>> add(4,5)
9
>>> r=add.delay(5,6)
[0] <3.9- 1:python*"be93d7f
```

It gives me the answer immediately, so in other words, I can use this directly as a function. But it becomes more interesting when I sort of say r is equal to let us say, add dot delay, and I give it 2 arguments. So, what is happening now? Now I am saying, do not execute it immediately. Push it onto the queue. And the moment I do this r is equal to add dot delay, I find that r comes back.

(Refer Slide Time: 11:13)

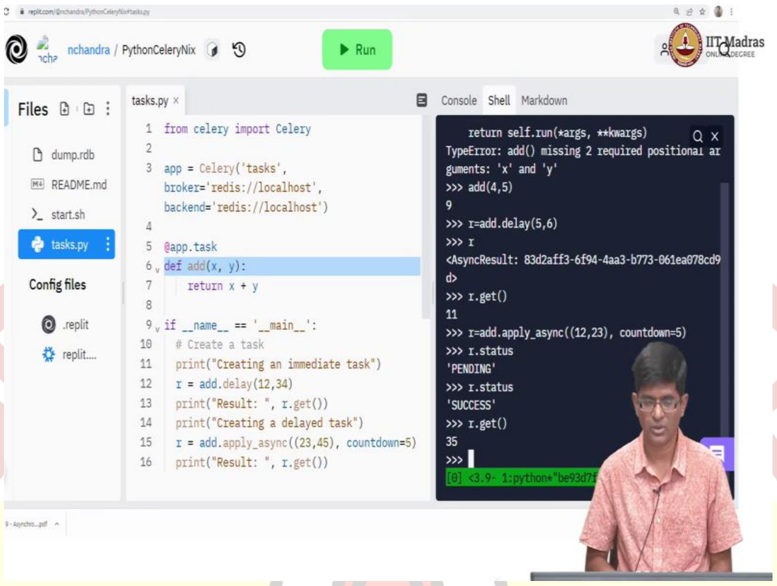


```python
from celery import Celery

app = Celery('tasks',
    broker='redis://localhost',
    backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
  print("Result: ", r.get())
```

```
[tasks]
  . tasks.add

[2022-02-15 05:54:36,469: INFO/MainProcess] Conne
cted to redis://localhost:6379//
[2022-02-15 05:54:36,557: INFO/MainProcess] mingl
e: searching for neighbors
[2022-02-15 05:54:37,645: INFO/MainProcess] mingl
e: all alone
[2022-02-15 05:54:37,662: INFO/MainProcess] celer
y@be93d7fbfce2 ready.
[2022-02-15 05:56:54,547: INFO/MainP        ] Task
tasks.add[83d2aff3-6f94-4aa3-b773-          9d] r
eceived
[2022-02-15 05:56:54,563: INFO/For          er-8]
Task tasks.add[83d2aff3-6f94-4aa3-b         ea078cd
9d] succeeded in 0.013485998000070
[0] 0:python3.9* 1>"be93d7f
```

I can go back to the shell, the dominant in which I was running the celery worker, and I find that there is some information out here. It says that pool worker has been a new pool worker has been forked. Fork is the term that is used in order to create a new thread. It ran tasks dot add. And it succeeded in well, 13 milliseconds. And the answer also has been received it is in fact, logged information. That is because I have given a high level of logging over here. The answer that came back from this function was the module level.

(Refer Slide Time: 11:53)



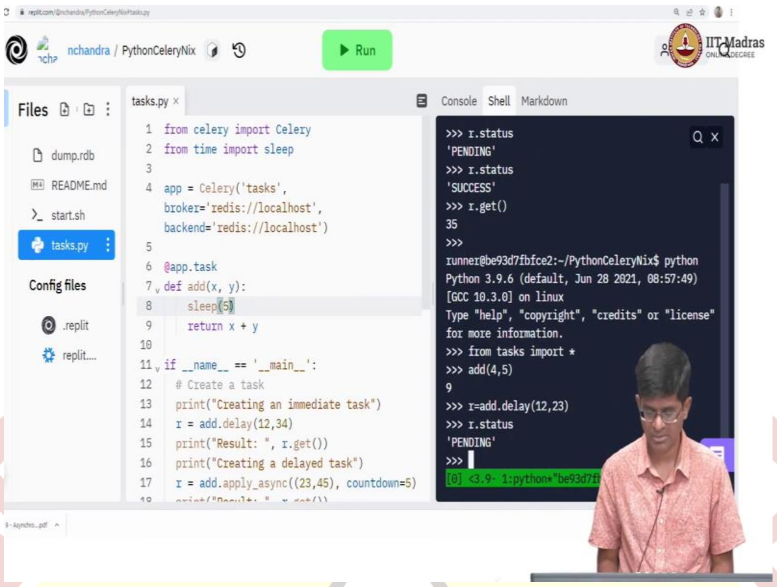
```python
from celery import Celery

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
  print("Result: ", r.get())
```

```
    return self.run(*args, **kwargs)
TypeError: add() missing 2 required positional ar
guments: 'x' and 'y'
>>> add(4,5)
9
>>> r=add.delay(5,6)
>>> r
<AsyncResult: 83d2aff3-6f94-4aa3-b773-061ea078cd9
d>
>>> r.get()
11
>>> r=add.apply_async((12,23), countdown=5)
>>> r.status
'PENDING'
>>> r.status
'SUCCESS'
>>> r.get()
35
>>>
[0] <3.9- 1:python*"be93d7f
```

But if I go back to my original terminal, the value level never got printed out. Why is that? Because if I look at r, r now has this result. But it does not show me the result by default. . I can get that result by doing r dot get at that point, it shows me yes, this was the result that r got. So, why was this useful? Obviously, like I said, for adding 2 numbers together, it is not useful. But let us say that this is a much more long running operation, I could have basically let it go to sleep and come back with this answer.

I can also do one more thing out here, which is I can explicitly tell the system to execute only after a certain amount of time. So I could, for example, say add dot call asynchronously, give it 2 parameters, and tell it to run only after 5 seconds. Now what happens r came back immediately. But now I can look at something called r dot status. And it says pending. Once again, look at r dot status. And after 5 seconds, it now shows success. So, instead of 5 seconds, if I had given 15 seconds, I would have been waiting along more to get a result back. And now when I get the results from r, I find that indeed, yes, it is 35.

(Refer Slide Time: 13:24)

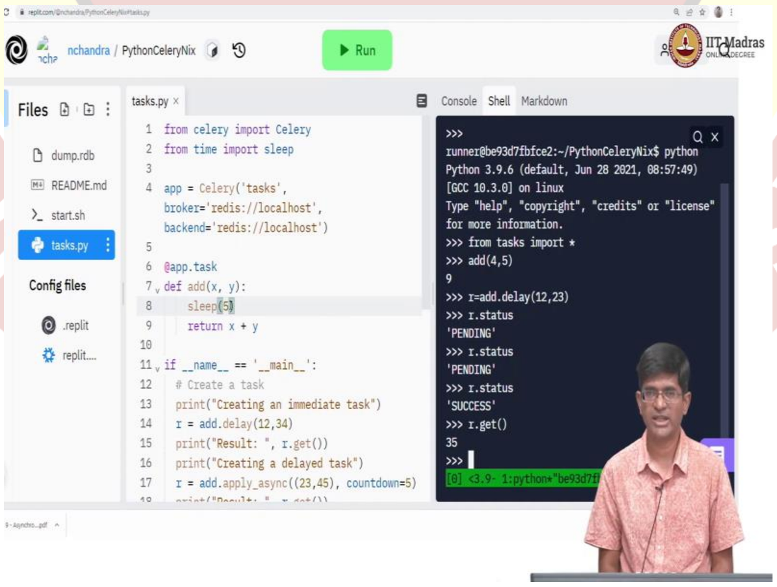


```python
from celery import Celery
from time import sleep

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    sleep(5)
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
```

```
>>> r.status
'PENDING'
>>> r.status
'SUCCESS'
>>> r.get()
35
>>>
runner@be93d7fbfce2:~/PythonCeleryNix$ python
Python 3.9.6 (default, Jun 28 2021, 08:57:49)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license"
for more information.
>>> from tasks import *
>>> add(4,5)
9
>>> r=add.delay(12,23)
>>> r.status
'PENDING'
>>>
```

I could also have just made this a long running operation. So, what I have done now is I have added a little bit more of a sleep operation into this task. And I continue doing the same way, from tasks, import star. And now what I would like to do is just called add, or, let us say that if I now directly, try doing add 4, 5, it just goes to sleep for 5 seconds. And eventually prints the result. Instead, what I could do is I could do ad dot delay 12, 23 and r comes back immediately. And I can now look at r dot status.

(Refer Slide Time: 14:14)



```python
from celery import Celery
from time import sleep

app = Celery('tasks',
broker='redis://localhost',
backend='redis://localhost')

@app.task
def add(x, y):
    sleep(5)
    return x + y

if __name__ == '__main__':
  # Create a task
  print("Creating an immediate task")
  r = add.delay(12,34)
  print("Result: ", r.get())
  print("Creating a delayed task")
  r = add.apply_async((23,45), countdown=5)
```

```
>>>
runner@be93d7fbfce2:~/PythonCeleryNix$ python
Python 3.9.6 (default, Jun 28 2021, 08:57:49)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license"
for more information.
>>> from tasks import *
>>> add(4,5)
9
>>> r=add.delay(12,23)
>>> r.status
'PENDING'
>>> r.status
'PENDING'
>>> r.status
'SUCCESS'
>>> r.get()
35
>>>
```

Pending, pending, success. And at that point, when I get the result, I find that indeed, it has got the correct answer. So, why did I put in that sleep 5 over there just to emulate a long running operation? In other words, if my function takes a long time to come back, all I am saying is that by using add delay over here, I have assigned it to the celery task queue. But the functionality immediately returns, so r comes back instantaneously.

Which means that I can now store that r value, the async result just some kind of ID corresponding to that result. All that I need to do is have it somewhere else. And eventually, I can look for the result that came back from it, I do not need to wait over there until it has actually finished execution.

So, this was a fairly simple example of how you can just use celery for running asynchronous tasks, obviously a very contrived example. But just for demonstration purposes, essentially. And, of course, as you can see, it allows you to use Redis, both for the backend as well as the broker. And the big, sort of slightly confusing part over here is the fact that when you want to use it on a platform like replit, you do have to sort of do something that is a bit non trivial use something like Nix in order to install these packages and so on.

On the other hand, hopefully, at this point, the most of you would have your own systems running with Linux or similar kind of setup where you can actually run all of this, you could actually have a Redis server, you could have a celery worker being created. And you could then run your own Python code directly on the system. The one catch is that celery does use the fork process in order to create new threads. What that means is, it does not natively work on something like Microsoft Windows.

And in order to get it running, you could use something like the windows subsystem for Linux, or there could be some other POSIX compatible environments that are there, that on which you might be able to run this. The preferred method would probably be either WSL, Windows subsystem for Linux or have a virtual machine within which you have Linux installed and get it working.

Now, why is this? Primarily because you will find that as you get into managing more complex servers with multiple different processes that need to interact with each other. The Linux kind of environment is generally speaking, easier to manage and more powerful in terms of what it gets you which is why we are sort of sticking to it here. And it is well worth knowing as well.