Custom Books (BDK)

An overview of the development kit used to create custom books.

In Kognitos, books can be added to an automation agent to expand functionality and integrate with external tools. Although Kognitos builds and maintains a **library** of books to address common use cases, there may be situations that require a **custom book**. Using our software toolkit, you can develop, deploy, and connect your own book to help with:

1. Integrations

Create procedures that interact with third-party APIs, databases, and external services. If your organization uses tools or services not natively integrated with Kognitos, you can build a custom book to integrate them directly into your automation workflows.

2. Extending Core Functionalities

Extend Kognitos' features to support more complex scenarios beyond standard functionality. Whether you need to support advanced data manipulation or custom data validation, the toolkit provides the tools to create custom books that meet your unique requirements.

Components

The BDK is comprised of the following software components:

	Component	Description
1	**API**	The core Python library for Book development.
2	**Template**	A Python Cookiecutter ↗ template that simplifies the process of developing a new book.
3	**Linter**	A linting tool built as a Pylint ↗ plugin. It analyzes code throughout development to identify potential issues and ensures books are implemented, documented, and annotated correctly.
4	**Poetry Plugin**	A Python dependency management and packaging tool that uses Poetry ↗. It manages dependencies, versioning, formatting, and documentation generation for a book.
5	**Runtime**	A runtime that connects a book with the Kognitos platform. It is packaged as a Docker image that serves as the foundation for book containers.

Lifecycle

This lifecycle outlines the stages of software development for a new book and how the components interact throughout the process.

1. Planning

The planning phase is where the book's functionality is defined. In this stage, you identify the external systems the book will integrate with and outline the workflows or automation goals. This stage clarifies objectives and requirements, setting the foundation for development.

2. Development

The development phase is where the book’s functionality is implemented using the **API** and other tools:

- The **Book Template** provides starter code to streamline setup.
- The **Linter** enforces coding standards and identifies potential issues.
- The **Poetry Plugin** manages dependencies, versioning, and documentation.

3. Testing

The testing phase ensures that books perform as intended. The **Template** includes a testing framework for validating book functionality.

4. Packaging & Deployment

In this stage, the book is packaged and prepared for deployment. The code is packaged with the **Runtime** using Docker and deployed externally.

5. Execution

Once deployed, the book is ready to run in Kognitos, enabling you to extend your automations with custom workflows.

Example Books

Book that integrates with the OpenWeather API.

Book that integrates with the Twilio API.

Book containing automation procedures that operate on YAML files.

Setup

Learn how to set up your custom book project.

Prerequisites

Ensure the following prerequisites are met and dependencies are installed before proceeding.

1. **Python 3.11+** ↗

<details>
<summary>pyenv *(recommended)*</summary>

Pyenv manages Python versions and project dependencies with the **pyenv-virtualenv** plugin.

1. Follow the installation steps for pyenv ↗.
2. Follow the installation steps for pyenv-virtualenv ↗.

Python can be installed with pyenv in the following way:

```
pyenv install 3.11
```
</details>

2. **Git** ↗

3. **Docker** ↗

4. **pipx** ↗

5. **Poetry** ↗

6. **Cookiecutter** ↗

Setting Up Your Project

Set up your project using the **Template**, a tool designed to simplify the creation of books.

1. Clone and Initialize

Clone the Template↗ from GitHub and initialize your project with Cookiecutter:

```
cookiecutter git+ssh://git@github.com/kognitos/bdk-template.git
```

Enter the following project details or press **Enter** to keep the defaults:

Configuration	Description	Default
Project Name	The name of your project.	Sample Book
Project Slug	A URL-friendly version of the project name. If not provided, this will be derived from the project name.	sample_book
Project Description	A short description of the project.	A short description of the project.
Author Name	You or your organization's name.	Kognitos
Initial Version	Initial project version.	0.1.0

2. Enter Project Directory

Navigate into the new project directory. Replace `project_slug` with your own **Project Slug** from the previous step:

```
cd <project_slug>
```

3. Create a Virtual Environment *(recommended)*

We recommend creating a virtual environment to isolate and manage your project dependencies.

1. Configure Poetry ↗ to create the virtual environment inside the project's root directory:

```
poetry config virtualenvs.in-project true
```

2. Create a virtual environment:

```
poetry env use 3.11
```

<details>
<summary>Setting your local pyenv version</summary>

If you are using pyenv and encounter the following error:

```
pyenv: python3.11: command not found

  The python3.11' command exists in these Python versions:
   3.11.11
```

Run the following command to set the local pyenv version for the current directory:

```
pyenv local 3.11
```

</details>

3. Activate the virtual environment

Run `poetry shell` ***or*** manually activate the environment ↗ using the following commands:

macOS/Linux

```
source $(poetry env info --path)/bin/activate
```

Windows (PowerShell)

```
Invoke-Expression (poetry env activate)
```

> **Note:** To later exit the virtual environment later, run `deactivate`.

4. Install dependencies

```
poetry install
```

Development

Learn how to develop, package, and run your book locally.

> **Have You Completed the Setup?**
>
> Ensure you've followed all the steps to set up your project before proceeding. This includes configuring your environment and installing dependencies.

Project Structure

The project is organized with the following structure:

```
sample_book
├── src
│   └── sample_book
│       ├── __init__.py
│       ├── __version__.py
│       └── book.py
│       └── data
│           └── icon.svg
├── tests
│   └── test_book.py
├── Dockerfile
├── pyproject.toml
├── poetry_scripts.py
├── poetry.lock
├── README.md
└── USAGE.md
```

Root Directory

The project root directory is named using the Project Slug *(default: sample_book)*.

Source Code Directory

- **`book.py`**: Contains the core class where your book is defined and implemented.

- `data`: Holds images like `icon.svg`, the default SVG used as a book's icon.

Test Directory

- `test_book.py`: Contains unit tests to validate the Book's functionality.

Configuration and Dependency Files

- `Dockerfile`: Builds the Docker image.
- `pyproject.toml`: Manages dependencies and settings.
- `poetry_scripts.py`: Custom automation script.
- `poetry.lock`: Locks dependency versions.

Documentation

- `README.md`: Project overview documentation, including setup and usage instructions.
- `USAGE.md`: Reference documentation for your Book. This file is not included by default. For details on how to generate it, see Generating Documentation.

Implementation

Implement your book in **book.py**.

This class is auto-generated by the Template and serves as the starting point for your Book's implementation. Use decorators to configure settings, define procedures, and establish API connections. Refer to the **API Reference** and **example books** ↗ for details.

Managing Dependencies

Poetry ↗ is used for Python dependency management in custom book projects. To add an external Python dependency to your project, refer to the **poetry add** ↗ command, which adds packages to your `pyproject.toml` and installs them.

Development Commands

This section outlines essential commands for book development and testing.

Formatting

Code formatting automatically adjusts your code's structure, ensuring adherence to coding standards, readability, and maintainability. To apply the formatting, run:

```
poetry run format
```

Linting

Pylint ↗ is used as the source linter for custom book projects. The linter analyzes and enforce coding standards, checks for errors in Python code, and improves code quality. To run the linter:

```
poetry run lint
```

Generating Documentation

This command will generate a comprehensive **USAGE.md** file by extracting and formatting the docstrings from your code. To generate the documentation, run:

```
poetry run doc
```

Testing

Pytest ↗ is used to test and validate your book. To run the test suite:

```
poetry run tests
```

Packaging with Docker

Build a Docker image to package your book:

```
docker build -t <project_slug>:<version> .
```

Architecture Compatibility

This image pulls the **Runtime** base image from Docker Hub, which is currently available only for the **linux/amd64** architecture. If you're using a machine with a different architecture, you can use the `--platform` flag to emulate linux/amd64 when building the image:

```
docker build --platform linux/amd64 -t <project_slug>:<version>
.
```

Running Your Book Locally

To test your Book in Kognitos before deployment, you can run your Docker image locally with ngrok ↗. Ngrok is a tool that creates a secure tunnel to your local machine and provides a public URL to make your local Docker image accessible from the platform.

1. Install ngrok

Follow the installation steps for **ngrok** ↗ based on your system. You will need to sign up for a free account.

2. Obtain your ngrok authtoken

Navigate to *Your Authtoken* in the ngrok portal and copy your token.

3. Add your authtoken

```
ngrok config add-authtoken <token>
```

4. Configure the ngrok api key as an environment variable:

```
export NGROK_AUTHTOKEN=<token>
```

5. Build & run in ngrok mode

This command will invoke a custom script that builds the Docker image and runs it in ngrok mode:

```
poetry run host
```

If you are running into issues with `poetry run host`, you can package and run your book manually with the following command:

```
docker run -e BDK_SERVER_MODE=ngrok -e NGROK_AUTHTOKEN=<auth_token>
<project_slug>:<version>
```

Make sure to replace `auth_token`, `project_slug`, and `version` with your own values.

6. Copy the URL

You'll see the ngrok address in the following format in the logs. Copy the URL:

```
listening on https://<SOME_UUID>.ngrok-free.app
```

7. Add the URL to your automation

Copy the URL and paste it into your automation using this syntax:

```
learn "https://f1d7-100-34-252-18.ngrok-free.app"
```

If you implemented any custom connections, make sure you also **connect** to your book *after* learning.

Deployment

Learn the requirements for deploying a custom book.

After successfully building your book and creating a Docker image, the next step is deployment. You have the flexibility to deploy your book to any environment that supports external access and routing, ensuring it can be reached from Kognitos. The choice of deployment method will depend on your organization's specific preferences and requirements. Common deployment options include:

- Cloud platforms (ex: Amazon ECS ↗, Azure ↗)
- On-premises infrastructure
- Container orchestration platforms, such as Kubernetes ↗

Port Configuration

Ensure that the port specified in the Dockerfile is properly exposed and matches your deployment environment's configuration.

API Reference

Concepts

Understand concepts and how they are utilized in procedures.

What are Concepts?

Concepts define how data flows in and out of a procedure:

- **Input concepts** represent the inputs that a procedure requires.
- **Output concepts** represent the results that a procedure generates.

Concept Types

Input and output concepts can be defined as either **standard** or **custom** types.

Standard Types

The following Python data types can be used to define concepts:

Category	Supported Python Data Types
Text	`str`
Number	`float`, `int`
Boolean	`bool`
Bytes	`bytes`
Date	`datetime.datetime`, `datetime.date`, `datetime.time`
File	`typing.IO`
UUID	`uuid.UUID`
Table	`pyarrow.Table`, `arro3.core.Table`, `nanoarrow.ArrayStream`
Lists	`List[str]`, `List[int]`
Dictionary	`Dict[str, Any]` > *Note: `Any` can be any of the other supported types.*

Custom Types

Concepts can also be **custom** types. Custom concepts must be marked with the **@concept** decorator. Using a dataclass↗ is recommended. For example:

```
@concept(is_a="office user")
@dataclass
class OfficeUser:
    """
    An Office User represents a user in the Microsoft Graph. It includes
key user details such as display name,
    email address, and job title.

    Attributes:
        id: The unique identifier for the user.
        display_name: The name displayed in the address book for the
user.
        email_address: The user's email address (usually their user
principal name).
        job_title: The user's job title.
    """

    id: str
    display_name: Optional[str] = None
    email_address: Optional[str] = None
    job_title: Optional[str] = None
    ...
```

Concepts vs. Parameters

Concepts and **parameters** are two different entities that are closely related.

Concepts

Concepts are operated on by Kognitos procedures. Some concepts are included in the **name** of the @procedure decorator.

Parameters

Parameters are operated on by Python functions. They are defined in a function's signature and are specific to the function's implementation.

How Are They Related?

An internal mapping is created between a Python function and a Kognitos procedure. To ensure a correct mapping, concepts and parameters must **match**.

Concept-Parameter Matching

Concepts and parameter names must **match** to ensure they are properly mapped internally.

Guidelines

Match concepts to parameters by following these guidelines:

1. **Replace spaces with underscores.**
2. **Drop possessive constructions** ('s).
3. **Consider each connected noun phrase** as a separate parameter. Non-essential noun phrases (e.g., "field," "value") included to clarify context do not require a corresponding parameter.
4. **Don't map articles** ('an', 'a', 'the') to parameters; only the nouns they connect should be considered.
5. **Define optional parameters** in cases where the input concept is not explicitly defined in the procedure name.

Example 1

In this example, the concept **red car** maps to the parameter `red_car`. The space is replaced with an underscore.

```
@procedure("to start a red car")
def unique_function_name(self, red_car: type) -> output_type:
```

Example 2

In the example below:

- **servicenow's ticket** maps to `ticket`
- **field** maps to `field`
- **outlook's standard user** maps to `standard_user`

```
@procedure("to send a servicenow's ticket's field to an outlook's
standard user)
def func(self, ticket: Ticket, field: NounPhrase, standard_user:
OutlookUser):
```

Example 3

In this example, the concept **ticket** maps to the parameter `ticket`. The possessive construct ('s) is dropped and the word "field" is ignored.

```
@procedure("to update a ticket's field")
def update_field(self, ticket: Ticket, new_value: Any):
```

Example 4

In the example below, the concept of **priority** is not explicitly stated in the procedure name. It maps to the optional function parameter, `priority`.

```
@procedure("to assign the task to the user")
def assign_task(self, task: Task, user: User, priority: Optional[str] =
None) -> None:
  """Assign a task to a user.

  Input Concepts:
    the task: The action of piece of work that needs to be completed.
    the user: The user that the action is to be assigned to.
    the priority: The level of importance or urgency of the task.

  Example 1:
    Assign the task to the user

    >>> assign the task to the user

  Example 2:
    Assign the task to the user with a priority

    >>> assign the task to the user with
          the priority is "high"
  """
```

Example 5

In this example:

- **the city** maps to the parameter `city`
- **the unit** maps to the optional parameter `the unit`

The output concept, **current temperature**, is wrapped in parentheses in the procedure name.

```
@procedure("to get the (current temperature) at a city")
def current_temperature(self, city: NounPhrase, unit:
Optional[NounPhrase] = NounPhrase("standard")) -> float:
    """Fetch the current temperature for a specified city.

    Input Concepts:
        the city: The name of the city.
        the unit: Unit of measurement.

    Output Concepts:
        the current temperature: The current temperature in the
specified units of measurement, or None if an error occurs.

    Example 1:
        Retrieve the current temperature at London

        >>> get the current temperature at London

    Example 2:
        Retrieve the current temperature at London in Celsius

        >>> get the current temperature at Buenos Aires with
        ...     the unit is metric
    """
```

Connections

Learn how to connect with third-party tools and services in your custom book project.

Implementing Connections

Connections enable you to connect to third-party tools and services in your custom Book. To implement a connection, define a Python function in your Book class and decorate it with the **@connect** decorator. This decorated function serves as the connection handler and defines the syntax for writing a connection command.

> **Note:** Implementing an **OAuth** connection in a custom BDK Book is not supported at this time.

Multiple Connections

You can define multiple connections within a Book, each with its own handler method. Use the `noun_phrase` keyword argument in the **@connect** decorator to assign a unique label to each connection and differentiate between authentication methods.

Method Docstrings

In your connection method docstring, include the following sections in addition to a brief summary:

1. Arguments

Specify the Python function's parameters.

2. Labels

Define labels for the connection arguments (e.g., API key, credentials) that will be used in your connection command. Labels correspond to parameters in your function's definition.

When a label is provided in the docstring, the lowercase form is used in a connection command. When *not* provided, the label will be inferred from the Python variable name.

Example

```
@connect(noun_phrase="api keys")
def connect(self, api_key: str):
    """
    Authenticate to Open Weather API using the specified API key. You
can obtain your own API key by visiting
    Open Weather's website at https://openweathermap.org/appid.

    Arguments:
        api_key: The API key to be used for connecting

    Labels:
        api_key: API Key
    """

    api_key = os.getenv("API_KEY", api_key)
    test_url = f"{self._base_url}?appid={api_key}&q=London"
    response = requests.get(test_url, timeout=self._timeout)
    if response.status_code == 401:
        response_data = response.json()
        if "Invalid API key" in response_data.get("message", ""):
            raise ValueError("Invalid API key")

    self._api_key = api_key
```

In this example, `API Key` is the label for the `api_key` connection argument. The lowercase version of the label is used in a connection command (`api key`):

```
connect openweather via some api keys method with
    the api key is "aBcDefg1234"
```

Additional Examples

1. `api_key: ApI kEy`

In this example, the connection command uses the lowercase form of `ApI kEy`:

```
connect openweather via some api keys method with
    the api key is "aBcDefg1234"
```

2. `api_key: aPi_KeY`

In this example, the connection command uses the lowercase version of `aPi_KeY`:

```
connect openweather via some api keys method with
    the api_key is "aBcDefg1234"
```

3. No Label

If a label is not provided in the docstring, it is inferred from the Python variable name, `api_key`:

```
connect openweather via some api keys method with
    the api key is "aBcDefg1234"
```

Docstrings

Understand how to format and write docstrings to ensure your Book is well-documented.

Format

BDK docstrings adhere to **Google style** ↗ formatting:

- Docstrings are placed **immediately after** function, class, or method definitions.
- A brief **summary** that describes the functionality begins the docstring.
- The summary is followed by organized **sections**, separated by blank lines and labeled with headers.

In BDK projects, docstrings are applied to decorated methods and classes.

Supported Sections

The following sections are supported in BDK docstrings, with alternative headers available for labeling flexibility.

1. Arguments and Parameters

Documents the arguments or parameters that a function or class accepts.

Supported Headers: `Args`, `Arguments`, `Parameters`, `Params`

Example

```
@procedure("to add two numbers")
def add_numbers(a: int, b: int = 0) -> int:
    """
    Adds two numbers together.

    Args:
        a: The first number.
        b: The second number. Defaults to 0.
    """
    return a + b
```

2. Return Values

Documents the return value of a function or method.

Supported Headers: `Returns`

Example

```
@procedure("to send an *SMS* message")
def send_sms_message(self, sender_number: str, recipient_number: str,
message_body: str) -> Optional[str]:
    """
    Sends an SMS message using the Twilio API.

    Returns:
        The SID of the sent message if successful, otherwise None.
    """
```

3. Error Handling

Lists any exceptions or errors that a function might raise.

Supported Headers: `Raises`, `Exceptions`, `Except`

Example

```
@timeout.setter
def timeout(self, timeout: float):
    """
    Sets the timeout value in milliseconds.

    Args:
        timeout (int): The timeout value to set. Must be a positive
integer.

    Raises:
        ValueError: If the timeout value is less than or equal to 0.

    """
    if timeout <= 0:
        raise ValueError("timeout must be positive")
    self._timeout = timeout
```

4. Instance Attributes

List instance attributes in a class.

Supported Headers: `Attributes`

Example

```
@concept(is_a="office user")
@dataclass
class OfficeUser:
    """
    An Office User represents a user in the Microsoft Graph. It includes
key user details such as display name,
    email address, and job title.

    Attributes:
        id: The unique identifier for the user.
        display_name: The name displayed in the address book for the
user.
        email_address: The user's email address (usually their user
principal name).
        job_title: The user's job title.
    """

    id: str
    display_name: Optional[str] = None
    email_address: Optional[str] = None
    job_title: Optional[str] = None
```

5. Code Authors

Attributes the author(s) of the code.

Supported Headers: `Author`

Example

```
@book(name="Sample Book")
class SampleBook:
    """
    A book shown as an example.

    Author:
      Kognitos
    """
```

6. Labels

Defines labeled data associated with a function related to connections.

Supported Headers: `Labels`

Example

```
@connect(noun_phrase="api keys")
def connect(self, api_key: str):
    """
    Authenticate to Open Weather API using the specified API key.

    Labels:
        api_key: API Key
    """
```

7. OAuth Labels

Describes OAuth labels for classes decorated with the @oauth decorator.

Supported Headers: `OAuth Labels`

Example

```
@oauth(
    id="oauth",
    provider=OAuthProvider.MICROSOFT,
    flows=[OAuthFlow.AUTHORIZATION_CODE, OAuthFlow.CLIENT_CREDENTIALS],

authorize_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2
/v2.0/authorize",

token_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2/v2.
0/token",
    scopes=[
        "https://graph.microsoft.com/Files.ReadWrite",
        "https://graph.microsoft.com/Sites.Manage.All",
        "https://graph.microsoft.com/Sites.ReadWrite.All",
        "https://graph.microsoft.com/User.ReadWrite.All",
        "https://graph.microsoft.com/Directory.ReadWrite.All",
        "https://graph.microsoft.com/Group.ReadWrite.All",
        "https://graph.microsoft.com/GroupMember.ReadWrite.All",
        "https://graph.microsoft.com/Mail.ReadWrite",
        "https://graph.microsoft.com/Mail.ReadWrite.Shared",
        "https://graph.microsoft.com/Mail.Send",
        "https://graph.microsoft.com/Calendars.ReadWrite",
    ],
)
@book(name="MicrosoftBook")
class BaseMicrosoftBook:
    """
    Base class for all Microsoft Books.

    OAuth Arguments:
        TENANT_ID: The tenant ID of the Azure AD directory.

    OAuth Labels:
        TENANT_ID: Tenant ID
    """
```

8. OAuth Arguments

Describes OAuth arguments for classes decorated with the @oauth decorator.

Supported Headers: `OAuth Arguments`

Example

```
@oauth(
    id="oauth",
    provider=OAuthProvider.MICROSOFT,
    flows=[OAuthFlow.AUTHORIZATION_CODE, OAuthFlow.CLIENT_CREDENTIALS],

authorize_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2
/v2.0/authorize",

token_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2/v2.
0/token",
    scopes=[
        "https://graph.microsoft.com/Files.ReadWrite",
        "https://graph.microsoft.com/Sites.Manage.All",
        "https://graph.microsoft.com/Sites.ReadWrite.All",
        "https://graph.microsoft.com/User.ReadWrite.All",
        "https://graph.microsoft.com/Directory.ReadWrite.All",
        "https://graph.microsoft.com/Group.ReadWrite.All",
        "https://graph.microsoft.com/GroupMember.ReadWrite.All",
        "https://graph.microsoft.com/Mail.ReadWrite",
        "https://graph.microsoft.com/Mail.ReadWrite.Shared",
        "https://graph.microsoft.com/Mail.Send",
        "https://graph.microsoft.com/Calendars.ReadWrite",
    ],
)
@book(name="MicrosoftBook")
class BaseMicrosoftBook:
    """
    Base class for all Microsoft Books.

    OAuth Arguments:
        TENANT_ID: The tenant ID of the Azure AD directory.

    OAuth Labels:
        TENANT_ID: Tenant ID
    """
```

9. Input Concepts

Describes the **input concepts** for a procedure.

Supported Headers: `Input Concepts`

Example

```
@procedure("to get the (current temperature) at a city")
def current_temperature(self, city: NounPhrase, unit:
Optional[NounPhrase] = NounPhrase("metric")) -> float:
    """
    Fetch the current temperature for a specified city.

    Input Concepts:
        the city: The name of the city. Please refer to ISO 3166 for the
state codes or country codes.
        the unit: Unit of measurement. standard, metric and imperial
units are available. If you do
            not specify the units, metric units will be applied by
default.
    """
```

10. Output Concepts

Describes the **output concepts** for a procedure.

Supported Headers: `Output Concepts`

Example

```
@procedure("to get the (current temperature) at a city")
def current_temperature(self, city: NounPhrase, unit:
Optional[NounPhrase] = NounPhrase("metric")) -> float:
    """
    Fetch the current temperature for a specified city.

    Output Concepts:
        the current temperature: The current temperature in the
specified units of measurement, or None if an error occurs.
    """
```

11. Examples

Outlines usage examples for a procedure.

Supported Headers: `Example`, `Examples`, `Example [1-5]`

Example

```
@procedure("to get the (current temperature) at a city")
def current_temperature(self, city: NounPhrase, unit:
Optional[NounPhrase] = NounPhrase("metric")) -> float:
    """
    Fetch the current temperature for a specified city.

    Example 1:
        Retrieve the current temperature at London

        >>> get the current temperature at London

    Example 2:
        Retrieve the current temperature at London in Celsius

        >>> get the current temperature at Buenos Aires with
        ...     the unit is metric
    """
```

Decorators

@book

Overview

The `@book` decorator is used to identify a class as a book.

Syntax

```
@book(*args, **kwargs)
class BookClass:
   """
    A sample book.

    Author:
      Kognitos
    """
    # Book implementation here
```

Keyword Arguments

Argument	Type	Description
`id`	`str`	A unique identifier for the Book.
`icon`	`str`	Path to an icon representing the Book. If not provided, a default icon will be used.
`name`	`str`	The name of the Book. If not provided, the name will be inferred from the class name.
`noun_phrase`	`str`	The book noun phrase.

Default Icon

If an `icon` is not specified, this will be used as the default.

The default icon for a BDK Book.

Example

```
@book(name="Twilio", icon="data/twilio.svg")
class TwilioBook:
    """
     The Twilio Book enables users to interact with the Twilio API.
     """
     ...
```

@concept

Overview

The `@concept` decorator designates a class as a **concept**, allowing it to be used as a custom data type for defining procedure inputs and outputs.

Syntax

```
@concept(*args, **kwargs)
```

Keyword Arguments

Argument	Types	Description
`is_a`	`str`, `List[str]`	The name of the custom concept to be referred to in the Kognitos platform.*
`unset`	`Any`	Specifies a default value to represent an unset state for the concept.

> *In the case of a **List**, each element would correspond to a noun phrase (a noun and its adjectives) in the hierarchy of possessives. For example:
>
> `is_a=["red dragon", "big house", "wooden door"]` = red dragons's big house's wooden door

Example

The following example demonstrates the usage of the `@concept` decorator to define the custom concept **sms message**:

```
@concept(is_a="sms message")
@dataclass
class SMSMessage:
    """
    An SMS (Short Message Service) message. Represents a text
communication sent over a cellular network,
    typically between mobile phones.

    Attributes:
        body: The text content of the message
        num_segments: The number of segments that make up the complete
message. SMS message bodies that exceed
            the [character limit]
(https://www.twilio.com/docs/glossary/what-sms-character-limit) are
segmented
            and charged as multiple messages. Note: For messages sent
via a Messaging Service, `num_segments` is
            initially `0`, since a sender hasn't yet been assigned
        sender_number: The sender's phone number (in [E.164]
(https://en.wikipedia.org/wiki/E.164) format),
            [alphanumeric sender ID]
(https://www.twilio.com/docs/sms/quickstart),
            [Wireless SIM]
(https://www.twilio.com/docs/iot/wireless/programmable-wireless-send-
machine-machine-sms-commands),
            [short code](https://www.twilio.com/en-
us/messaging/channels/sms/short-codes), or
            [channel address]
(https://www.twilio.com/docs/messaging/channels) (e.g.,
`whatsapp:+15554449999`). For
            incoming messages, this is the number or channel address of
the sender. For outgoing messages, this value
            is a Twilio phone number, alphanumeric sender ID, short
code, or channel address from which the
            message is sent
        recipient_number: The recipient's phone number (in [E.164]
(https://en.wikipedia.org/wiki/E.164) format) or
            [channel address]
(https://www.twilio.com/docs/messaging/channels) (e.g.
`whatsapp:+15552229999`)
        date_updated: The [RFC 2822]
(https://datatracker.ietf.org/doc/html/rfc2822#section-3.3) timestamp
(in GMT) of
            when the Message resource was last updated
        price: The amount billed for the message in the currency
specified by `price_unit`. The `price` is populated
            after the message has been sent/received, and may not be
immediately availalble. View
```

```
            the [Pricing page](https://www.twilio.com/en-us/pricing) for
more details.
        account_sid: The SID of the [Account]
(https://www.twilio.com/docs/iam/api/account) associated with
            the Message resource
        num_media: The number of media files associated with the Message
resource.
        status: The status of the message, for more information about
possible statuses see
            [Message Status]
(https://www.twilio.com/docs/messaging/api/message-resource#message-
status-values)
        messaging_service_sid: The SID of the [Messaging Service]
(https://www.twilio.com/docs/messaging/api/service-resource)
            associated with the Message resource. A unique default value
is assigned if a Messaging Service is not used.
        sid: The unique, Twilio-provided string that identifies the
Message resource.
        date_sent: The [RFC 2822]
(https://datatracker.ietf.org/doc/html/rfc2822#section-3.3) timestamp
(in GMT) of when
            the Message was sent. For an outgoing message, this is when
Twilio sent the message. For an
             incoming message, this is when Twilio sent the HTTP request
to your incoming message webhook URL.
        date_created: The [RFC 2822]
(https://datatracker.ietf.org/doc/html/rfc2822#section-3.3) timestamp
(in GMT) of
            when the Message resource was created
        price_unit: The currency in which `price` is measured, in
            [ISO 4127]
(https://www.iso.org/iso/home/standards/currency_codes.htm) format (e.g.
`usd`, `eur`, `jpy`).
        error_message: The description of the `error_code` if the
Message `status` is `failed` or `undelivered`. If no
            error was encountered, the value is `null`.
        error_code: The [error code]
(https://www.twilio.com/docs/api/errors) returned if the Message
`status` is
            `failed` or `undelivered`. If no error was encountered, the
value is `null`.
    """

    sid: Optional[str]
    body: Optional[str]
    num_segments: Optional[str]
    sender_number: Optional[str]
    recipient_number: Optional[str]
```

@config

Overview

The `config` decorator is used to decorate a property in a Book class that defines default configuration values.

Syntax

```
@config(*args, **kwargs)
```

Keyword Arguments

Argument	Type	Required	Description
`name`	`str`	Optional	Specifies the name of the configuration. If not provided, the function name or value from `*args` is used.
`default_value`	`Any`	Optional	Specifies the default value for the configuration. If not provided, the default is `None`.

Example

```
DEFAULT_TIMEOUT = 30

@property
@config(default_value=DEFAULT_TIMEOUT)
def timeout(self) -> float:
 """
 Timeout in seconds when making API calls.
 """
 return self._timeout

@timeout.setter
def timeout(self, timeout: float):
 """
 Sets the timeout value in seconds.
 """
 if timeout <= 0:
  raise ValueError("timeout must be positive")
 self._timeout = timeout
```

@connect

Overview

The `@connect` decorator is used to wrap functions that handle **connections** or authentication tasks. It defines the syntax for writing a connection command, which instantiates the connection in Kognitos to the third-party API or service that's implemented in your Python function.

Syntax

```
@connect(*args, **kwargs)
```

Keyword Arguments

Argument	Type	Description
`name`	`str`	A name to associate with the connection. If not provided, it defaults to the function name.
`noun_phrase`	`str`	A unique label to identify the authentication method. If not provided, it will be inferred from the function name. Examples include, but are not limited to: • `noun_phrase="api keys"` • `noun_phrase="client credentials"` • `noun_phrase="user password authentication"`

Implementation Example

This is an example implementation of connecting to the OpenWeather API using an API key:

```
@connect(noun_phrase="api keys")
def connect(self, api_key: str):
  """
  Authenticate to Open Weather API using the specified API key.
  You can obtain your own API key by visiting Open Weather's
  website at https://openweathermap.org/appid.

  Arguments:
      api_key: The API key to be used for connecting

  Labels:
      api_key: API Key
  """
  api_key = os.getenv("API_KEY", api_key)
  test_url = f"{self._base_url}?appid={api_key}&q=London"
  response = requests.get(test_url, timeout=self._timeout)
  if response.status_code == 401:
      response_data = response.json()
      if "Invalid API key" in response_data.get("message", ""):
          raise ValueError("Invalid API key")

  self._api_key = api_key
```

Refer to **connections** for additional context and usage examples.

@oauth

Overview

The `@oauth` decorator is used to apply OAuth-based authentication to classes. It adds OAuth-specific metadata, such as endpoints, arguments, and provider details, to the class, enabling integration with OAuth services.

Syntax

```
@oauth(
    id=str,
    provider=OAuthProvider,
    flows=Optional[List[OAuthFlow]] = None,
    authorize_endpoint=str,
    token_endpoint=str,
    scopes=Optional[List[str]] = None,
)
class ClassName:
    """
    Class description
    """
    # Class implementation
```

Parameters

Parameter	Type	Required	Description
`id`	str	Yes	The unique identifier for the OAuth configuration.
`provider`	OAuthProvider	Yes	The OAuth provider: • OAuthProvider.MICROSOFT • OAuthProvider.GOOGLE
`flows`	List[OAuthFlow]	Optional	A list of OAuth flows.
`authorize_endpoint`	str	Yes	The authorization endpoint URL for the OAuth provider.
`token_endpoint`	str	Yes	The token endpoint URL for exchanging the authorization code for a token.
`scopes`	List[str]	Optional	A list of scopes required by the OAuth provider for the authenticated requests.

Example

```
@oauth(
    id="oauth",
    provider=OAuthProvider.MICROSOFT,
    flows=[OAuthFlow.AUTHORIZATION_CODE],

authorize_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2
/v2.0/authorize",

token_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2/v2.
0/token",
    scopes=[
        "https://graph.microsoft.com/Files.ReadWrite",
        "https://graph.microsoft.com/Sites.ReadWrite.All",
    ],
)
@book(name="MicrosoftBook")
class BaseMicrosoftBook
```

@oauthtoken

Overview

The `@oauthtoken` decorator marks a function as handling OAuth tokens, enabling the function to be automatically recognized as part of an OAuth workflow.

Syntax

```
@oauthtoken
def function_name(self, access_token: str, expires_in: Optional[int] =
None) -> None:
```

> **Note:** This decorator always applies to the same method signature, but the method itself can have any name.

Example

```
@oauth(
    id="oauth",
    provider=OAuthProvider.MICROSOFT,
    flows=[OAuthFlow.AUTHORIZATION_CODE],

authorize_endpoint="https://login.microsoftonline.com/{TENANT_ID}/oauth2
/v2.0/authorize"
```

@procedure

Overview

The `@procedure` decorator is used to denote a method within a Book class as a procedure. This links a method to a specific procedure in the Kognitos platform.

Syntax

```
@procedure(name: str, **kwargs)
```

Guidelines

1. Naming Conventions

Names must begin with **to**. This defines the action or intent of the procedure. For example:

- `@procedure("to capitalize a (string)"`
- `@procedure("to get the (current temperature)")`
- `@procedure("to send an *SMS* message")`

2. Output Concepts

Output concepts are wrapped in parentheses `()`. For example:

```
@procedure("to capitalize a (string)")
```

3. Proper Nouns

Proper nouns are wrapped between asterisks `**`. For example:

```
@procedure("to get some (users) from *office365*")
```

In this example, `office365` is considered a proper noun. The procedure is referred to as 'get some users from office365' rather than `the office365`.

Parameters

Parameter	Type	Required	Description
`name`	`str`	Yes	A description that reflects the action or purpose of the procedure. See the syntax guidelines for details.

Keyword Arguments

Argument	Type	Description
`connection_required`	`bool`	A boolean that indicates whether a connection to the service is required to execute the procedure. If not specified, it defaults to `None`.
`noun_phrase`	`str`	A string that represents the noun phrase for the procedure.

Examples

1. Capitalizing a String

This method implements a procedure that capitalizes a string with one input concept and one output concept.

```
@procedure("to capitalize a (string)", connection_required=False)
def capitalize_string(self, string: str) -> str:
  """
  Capitalizes the input string.

  Input Concepts:
    the string: The string value you want to capitalize.

  Output Concepts:
    the string: The capitalized string.

  Example 1:
    Capitalize the string "hello"

    >>> capitalize "hello"

  """
  return string.capitalize()
```

2. Creating an Order in Truckmate

This method implements a procedure that creates an order in Truckmate. It has one input concept and two output concepts.

```
@procedure("to create an order in truckmate and get the order number and
the bill number")
def create_order(self, order: OrderRequestConcept) -> Tuple[int, str]:
    """
    Create a new order in Truckmate.

    Input Concepts:
        the order: The Truckmate order request (acc to openAPI spec)

    Output Concepts:
        the order number: Order ID of the created order
        the bill number: Bill Number of the created order

    Raises:
        ValueError: If the order is not created successfully.

    Example 1:
        Create the order in Truckmate
        >>> create a json
        >>> use the above as the order
        >>> set the order's id to 1234
        >>> set the order's title to "Awesome title"
        >>> set order's body to "This is the body"
        >>> create the order in truckmate and get the order number and
bill number
    """
    logger.info("Creating Order in Truckmate")
    post_body = OrdersPostRequest(orders=[order])
    response = post_orders.sync_detailed(
        client=self._client,
        body=post_body,
    )
    if response.status_code == 201:
        orders_response = response.parsed.to_dict()  # type: ignore
        if orders_response.get("orders"):
            order = orders_response.get("orders")[0]  # type: ignore
            return order.get("orderId"), order.get("billNumber")

    error_response = response.parsed.to_dict()  # type: ignore
    logger.error(
        "Error while creating order in truckmate:  %s",
        error_response.get("errorText"),
    )
    raise ValueError(error_response)
```

3. Reading SMS messages using the Twilio API

The following method implements a procedure that reads SMS messages using the Twilio API. In this example, `SMS` is a proper noun.

```
@procedure("to read some (*SMS* messages)")
def read_sms_messages(
    self,
    offset: Optional[int],
    limit: Optional[int],
    filter_expression: Optional[FilterExpression],
) -> List[SMSMessage]:
    """
    Read some SMS messages using the Twilio API.

    Returns:
        A list of SMS messages that matches the specified filtering
criteria

    Example 1:
```

Enums

FilterBinaryOperator

Enum Definition

```
from enum import Enum

class FilterBinaryOperator(Enum):
    """Represents the different binary operators used in filtering
procedures."""
    AND = 0
    OR = 1
    EQUALS = 2
    NOT_EQUALS = 3
    IN = 4
    HAS = 5
    LESS_THAN = 6
    GREATER_THAN = 7
    LESS_THAN_OR_EQUAL = 8
    GREATER_THAN_OR_EQUAL = 9
```

Enum Members

Member	Value	Description
`AND`	`0`	Logical AND operator, used to combine conditions.
`OR`	`1`	Logical OR operator, used to combine alternative conditions.
`EQUALS`	`2`	Tests if two values are equal.
`NOT_EQUALS`	`3`	Tests if two values are not equal.
`IN`	`4`	Tests if a value is within a specified list or range.
`HAS`	`5`	Tests if a collection contains a specific element.
`LESS_THAN`	`6`	Tests if one value is less than another.
`GREATER_THAN`	`7`	Tests if one value is greater than another.
`LESS_THAN_OR_EQUAL`	`8`	Tests if one value is less than or equal to another.
`GREATER_THAN_OR_EQUAL`	`9`	Tests if one value is greater than or equal to another.

FilterUnaryOperator

Enum Definition

```
from enum import Enum

class FilterUnaryOperator(Enum):
    """Represents the unary operators used in filtering procedures."""
    NOT = 0
```

Enum Members

Member	Value	Description
`NOT`	`0`	Logical NOT operator, negates a condition.

Filter Expressions

Learn about implementing filter expressions in your custom book project.

Overview

Filter expressions enable you to define filter criteria for your procedures using the **whose** keyword. These expressions allow you to filter data based on conditions such as equality, comparisons, and more. For example:

```
get users from office365 whose email is "john@mail.org"
```

Implementation

1. Include the `filter_expression` parameter

To implement a filter expression, you need to provide the special `filter_expression` parameter in your procedure method definition. For example:

```
@procedure("to read some (*SMS* messages)")
def read_sms_messages(
    self,
    offset: Optional[int],
    limit: Optional[int],
    filter_expression: Optional[FilterExpression],
) -> List[SMSMessage]:
    """
    Read some SMS messages using the Twilio API.
    """
```

2. Implement Visitors

The `FilterExpressionVisitor` class is an abstract base class that defines methods for visiting different types of filter expressions. Each type of filter expression (binary, unary, value, noun phrase) is defined as a subclass of `FilterExpression`. You will need to define a class that implements these methods to handle filter expressions. For example:

```
class MyFilterVisitor(FilterExpressionVisitor):
    def visit_binary_expression(self, expression:
FilterBinaryExpression) -> T:
        # Implement logic for handling binary expressions
        pass

    def visit_unary_expression(self, expression: FilterUnaryExpression)
-> T:
        # Implement logic for handling unary expressions
        pass

    def visit_value(self, expression: ValueExpression) -> T:
        # Implement logic for handling value expressions
        pass

    def visit_noun_phrases(self, expression: NounPhrasesExpression) ->
T:
        # Implement logic for handling noun phrases expressions
        pass
```

Below is an example implementation of the `FilterExpressionVisitor` class in the Twilio book:

```
SENDER_NUMBER = NounPhrase.from_word_list(["sender", "number"])
RECIPIENT_NUMBER = NounPhrase.from_word_list(["recipient", "number"])
DATE_SENT = NounPhrase.from_word_list(["date", "sent"])


class SMSMessageFilter(FilterExpressionVisitor):
    """
    Extract filtering information from the expression
    """

    current_noun_phrase: Optional[NounPhrase] = None
    current_value: Optional[Any] = None

    recipient_number: Union[str, object] = values.unset
    sender_number: Union[str, object] = values.unset
    date_sent: Union[datetime, object] = values.unset
    date_sent_before: Union[datetime, object] = values.unset
    date_sent_after: Union[datetime, object] = values.unset

    def visit_binary_expression(self, expression:
FilterBinaryExpression):
        expression.left.accept(self)
        expression.right.accept(self)

        if expression.operator == FilterBinaryOperator.EQUALS:
            if self.current_noun_phrase == SENDER_NUMBER:
                self.sender_number = str(self.current_value)
            elif self.current_noun_phrase == RECIPIENT_NUMBER:
                self.recipient_number = str(self.current_value)
            elif self.current_noun_phrase == DATE_SENT:
                if not isinstance(self.current_value, datetime):
                    raise TypeMismatchError("date_sent"
```

3. Processing Filter Expressions

Once you've defined your visitor class, you need to pass an instance of it to the filter expression. This is done by calling **accept** on `filter_expression`. For example:

```
visitor = MyFilterVisitor()
filter_expression.accept(visitor)
```

Example

In this example, a filter expression is used in the **read some SMS messages** procedure:

```
@procedure("to read some (*SMS* messages)")
def read_sms_messages(
    self,
    offset: Optional[int],
    limit: Optional[int],
    filter_expression: Optional[FilterExpression],
) -> List[SMSMessage]:
    """
    Read some SMS messages using the Twilio API.

    Returns:
        A list of SMS messages that matches the specified filtering
criteria

    Example 1:
```

Noun Phrases

Learn about the concept of a noun phrase.

Overview

A **noun phrase** is a group of words centered around a noun. It consists of:

- **Head:** The main noun representing the fact.
- **Modifiers:** Optional words providing additional context to the head noun.

The `NounPhase` class can be used to represent a noun phrase.

Modifiers

A **modifier** is an optional part of a noun phrase that provides additional detail about the **head** noun. It appears before the head and refines the fact being referenced.

Example: Creating a Noun Phrase with Modifiers

```
# Create a noun phrase with a head and modifiers
np = NounPhrase("dog", ["big", "white"])
print(np)  # Outputs: big white dog
```

In this example:

- **Head:** "dog"
- **Modifiers:** "big", "white"

The resulting noun phrase is "big white dog".

Parameters as Noun Phrases

Parameters can be defined as `NounPhrase` types to tell the system to use a fact's **name** instead of its **value**.

Example: Defining a Parameter as a Noun Phrase

Consider the following procedure method definition, where the `city` parameter is defined as a `NounPhrase`:

```
@procedure("to get the (current temperature) at a city")
def current_temperature(
  self, city: NounPhrase, unit: Optional[NounPhrase] =
NounPhrase("metric")
) -> float:
```

This procedure can be called in an automation where **London** is a fact with a specific value:

```
London is "windy"
get the current temperature at London
```

In this example, when **London** is specified as the city parameter, the system uses the name of the fact (London) instead of its value ("windy").

Procedures

Learn how to implement a procedure in a custom book.

To implement a procedure in a custom book, define a Python function in your book class and decorate it with **@procedure**.

Requirements

1. Naming

Ensure the name of your procedure adheres to the syntax guidelines specified in the `name` parameter of the @procedure decorator.

2. Method Docstrings

Your method docstring should include the following sections:

- A brief summary of the procedure
- Input Concepts
- Output Concepts

Examples are not required but are valuable for generating usage documentation.

3. Concept-Parameter Matching

Concepts and parameters must **match** to ensure they are properly mapped internally. Ensure your method definition adheres to the guidelines.

Using Procedures in Your Automation

Singularized Calls

A **singularized call** is a way to call a procedure by phrasing it as if it returns a single item, even though it returns a list by definition. A procedure supports singularized calls in addition to standard calls if it meets ***all*** of the following conditions:

- **Returns a list**.
- The **output** of the procedure is the object itself.
- The **output noun phrase** is plural.
- The procedure **accepts filters.**

> You don't need to implement additional logic for singular calls. The system will automatically generate the singularized variation of any procedure that meets the above criteria.

Example

Consider a procedure that retrieves users from Outlook. It can be called in two ways:

- **Standard Way**: `get some users from outlook whose whose mail is "example.com"`
- **Singularized Way**: `get a user from outlook whose whose mail is "example.com"`

```
@procedure("to get some (users) from *office365*")
async def retrieve_users(self, filter_expression:
Optional[FilterExpression] = None) -> List[OfficeUser]:
  """
  Retrieves Office 365 users accessible via the Microsoft Graph API.

  It requires the following permissions on the application:
  User.Read.All, User.ReadWrite.All, Directory.Read.All,
Directory.ReadWrite.All

  Returns:
  A list of Office 365 users.

  Example 1:
  Retrieve all users

  >>> get users from office365

  Example 2:
  Retrieve a user whose email matches the specified email address

  >>> get users from office365 whose mail is "john@acme.org"
  """
```

Questions

Learn how to implement questions in a custom book.

Overview

In your Book, you may need to **ask the user for information**. For example, if the book needs to know when a project should begin, it can ask the user, “What is the project start date?” Or if it needs a username, it can prompt the user with, “What is the username?”

When this happens, Kognitos surfaces a **question** to pause the automation and ask the user for information. A question is an exception that signals that the automation cannot move forward without user input. The toolkit provides an easy way to add and handle questions in your Book using the **`ask()`** method.

How It Works

The `ask()` function is used to **ask a question**. Here’s how it works:

1. **ask() is Called**

   `ask()` is called inside a procedure to ask a question.

2. **Kognitos Checks for an Answer**

   Each time `ask()` is called, Kognitos checks internally whether it has already stored an answer for that specific question before.

   - If the answer to the question **exists**, `ask()` returns the answer value itself.
   - If the answer to the question **does not exist**, `ask()` returns a `Question`.

3 Question is Surfaced *(if the answer is not present)*

When a `Question` object is returned from a procedure, Kognitos will:

1. Forward the question to the user
2. Pause the automation and wait for a response
3. Store the answer internally
4. Restart the procedure from the beginning automatically
    - *The second time the procedure runs, `ask()` will be able to return the answer instead of a Question.*

> **Note:** Kognitos only surfaces a question if it doesn't have the answer stored internally. By remembering answers, it ensures the same questions aren't unnecessarily presented to the user.

Usage

```
ask(concept_name: NounPhrasesStringLiteral, concept_type: Type[T], text:
Optional[str] = None, choices: Optional[List[T]] = None) -> Union[T,
Question[NounPhrasesStringLiteral, T]]
```

Parameter	Required	Description
`concept_name`	Yes	The unique name of the concept being asked for.
`concept_type`	Yes	The return type of the answer.
`text`	No	*(Optional)* A message to show the user when the question is prompted in Kognitos.
`choices`	No	*(Optional)* A list of predefined options the user can choose from.

To use `ask()` in your Book, follow these steps:

1 Import

Start by importing the `ask()` function and `Question` type from the API:

```
from kognitos.bdk.api.questions import ask, Question
```

2 Call ask()

Call the `ask()` function inside a method decorated with `@procedure`.

Provide the `concept_name` *(the unique name of the concept being asked for)* and the `concept_type` *(the return type of the answer)*.

```
result = ask("project start date", str)
```

(Optional) Specifying the Question Text

You can specify a `text` to present to the user when the question is prompted in Kognitos. If the text is not specified, questions are presented as `Please provide the <concept_name>` by default. For example:

```
result = ask("project start date", str, text="What is the start
date of the project?")
```

(Optional) Specifying Choices

You can also guide users to select an answer from a predefined list by passing a `choices` argument. When choices are set, Kognitos displays the question with a dropdown menu for the user to select from. For example:

```
result = ask("project start date", str, choices=["2025-07-22",
"2025-07-25", "2025-07-28"])
```

3 Handle the Result of ask()

The `ask()` function can return either **the answer** or a **`Question`** object. To handle both cases, explicitly check the return type:

- If `ask()` returns a value, that means the answer already **exists** internally.
- If `ask()` returns a `Question`, that means the answer **does not exist** internally. What you do with the `Question` is up to you — the Book can return it, hold onto it for later, or ignore it entirely. If you want Kognitos to surface the question to the user, return the `Question` object from the procedure. For example:

```
if isinstance(result := ask("project start date", str), Question)
    return result
```

4 Use the Answer

Once the `ask()` function returns a value *(and not a `Question`)*, you can use the answer just like any other variable in Python — in calculations, conditions, etc. For example:

```
if isinstance(start_date := ask("project start date", str),
Question):
    return start_date

# Use the answer as a datetime object
date_obj = datetime.strptime(date_str, "%Y-%m-%d")
project_due_date = date_obj + timedelta(days=30)
```

5 Declare Return Types

Always include **return types** in your procedure definitions. If your procedure may trigger a `Question`, the return type must reflect that so the system understands which concept(s) you may be asking for. Make sure to include both the **concept name** (as a string `Literal` in nounphrase form) and the **concept type** in the type hint. For example:

```
@procedure("to get a project due (date)")
def get_due_date(self) -> str | Question[Literal["project start
date"], str]:
```

Examples

1. Asking a Question with Different Return Types

This example shows how to use `ask()` with a set of mixed-type choices: a string, a datetime, and a float. The return type allows for any of those types, and also accounts for the case where a `Question` is returned.

```
@procedure("to get a (value)")
def get_value(self) -> str | datetime | float |
Question[Literal["value"], str | datetime | float]:

    choices = ["Hello!", datetime(2023, 1, 1, tzinfo=timezone.utc),
123.456]
    if isinstance(answer := ask("value", Union[str, datetime, float],
choices=choices), Question):
        return answer
    return answer
```

2. Asking Multiple Questions

Sometimes your book may need to collect more than one piece of information. You can call `ask()` multiple times in the same procedure. Here's an example:

```
from typing import Literal
from kognitos import procedure
from kognitos.bdk.api.questions import ask, Question

@procedure("to get a (text)")
def get_text(self) -> str | Question[Literal["color"], str] |
Question[Literal["elephant's name"], str]:
    """
    Get a text
    """
    # First: Ask the user to pick a color
    if isinstance(answer := ask("color", str, text="Pick a color",
choices=["red", "blue", "green"]), Question):
        return answer
    color = answer

    # Second: Ask for the elephant's name
    if isinstance(answer := ask("elephant's name", str, text="What is
the elephant's name?"), Question):
        return answer
    name = answer

    # You can now use both answers
    return f"The elephant named {name} is painted {color}."
```

How This Works

1. The first call to `ask()` checks whether the answer for **color** is present.
    a. If not, it returns a Question.
    b. If answered, it stores the value and restarts the procedure.
2. Now that the answer for color is present, the procedure checks for the **elephant's name**.
    a. If not, it returns a Question.
    b. If answered, it stores the value and restarts the procedure.
3. Once both values are available, the final result is returned.