ME
DECON
MATERIAL DE ENSEÑANZA

N° 8

ECONOMÍA EXPERIMENTAL APLICADA. PROGRAMACIÓN DE EXPERIMENTOS CON OTREE

Ricardo Huamán-Aguilar

Departamento
Académico de Economía

PUCP

MATERIAL DE ENSEÑANZA N° 8

Economía Experimental Aplicada. Programación de experimentos con oTree

Ricardo Huamán-Aguilar

Febrero, 2023

MATERIAL DE ENSEÑANZA 8
http://files.pucp.edu.pe/departamento/economia/ME008.pdf

Economía Experimental Aplicada. Programación de experimentos con oTree

Editado
© Departamento de Economía – Pontificia Universidad Católica del Perú
Av. Universitaria 1801, Lima 32 – Perú.
Teléfono: (51-1) 626-2000 anexos 4950 - 4951
econo@pucp.edu.pe
https://departamento.pucp.edu.pe/economia/publicaciones/materiales/

Encargada de la Serie: Janina V. León Castillo
Departamento de Economía – Pontificia Universidad Católica del Perú,
jaleon@pucp.edu.pe

Economía Experimental Aplicada

Programación de experimentos con oTree

Ricardo Huamán-Aguilar

con la colaboración de

Úrsula Cotrina
Nicole Linares
Andrea Ulloa

LEEX - PUCP
LABORATORIO DE ECONOMÍA EXPERIMENTAL

Departamento de Economía

PUCP

21 de febrero de 2023

Economía Experimental Aplicada.
Programación de experimentos con oTree

Ricardo Huamán-Aguilar
con la colaboración de:
Úrsula Cotrina
Nicole Linares
Andrea Ulloa

RESUMEN

El presente documento tiene su origen en el material preparado para el Taller Básico de Economía Experimental, organizado por LEEX-PUCP en julio del año 2022. Tiene como objetivo principal presentar los elementos básicos para programar experimentos en la plataforma oTree. Está dirigido a estudiantes, investigadores y profesionales interesados en hacer investigación en el área experimental. Es un documento autocontenido y, en principio, no tiene prerrequisitos. No obstante, tener conocimientos previos de teoría de juegos, y de algún lenguaje de programación, facilita la lectura.

Además de la Introducción, este documento se divide en 4 capítulos y un apéndice. El capítulo 1 es una revisión breve de la economía experimental. Se presentan los conceptos básicos, los elementos de un experimentos y los tipos de experimentos. Dado que oTree es una plataforma basada en Python, en el capítulo 2 se presentan los conocimientos mínimos de dicho programa. Los capítulos 3 y 4 tratan de la programación de experimentos en oTree. En el capítulo 3 se presentan los elementos básicos, usando como ejemplo la programación de una encuesta predeterminada en oTree. En el capítulo 4 se profundiza en la programación de experimentos, usando como ejemplo el juego de los bienes públicos predeterminado en oTree. Finalmente, el apéndice presenta los complementos necesarios.

Palabras clave: economía experimental, oTree, programación de experimentos, experimentos de laboratorio, experimentos de campo, experimentos naturales.

Código JEL: C90, C91, C92, C93

ABSTRACT

This document originates from the material prepared for the Basic Workshop on Experimental Economics, organized by LEEX-PUCP in July 2022. Its main objective is to present the basic elements for programming experiments on the oTree platform. It is aimed at students, researchers and professionals interested in doing research in the experimental area. It is a self-contained document and, in principle, has no prerequisites. However, having previous knowledge of game theory, and of some programming language, makes reading easier.

In addition to the Introduction, this document is divided into 4 chapters and an appendix. Chapter 1 is a brief review of experimental economics. The basic concepts, the elements of an experiment and the types of experiments are presented. Since oTree is a Python-based platform, Chapter 2 introduces the basic knowledge of such a program. Chapters 3 and 4 deal with programming experiments in oTree. Chapter 3 introduces the basic elements, using the default survey in oTree as an example. In Chapter 4, we go deeper into programming experiments, using the public goods game set in oTree. Finally, the appendix presents the necessary complementary material.

Keywords: experimental economics, oTree, programming experiments, Lab experiments, field experiments, natural experiments.

JEL code: C90, C91, C92, C93

Índice general

Introducción

En las últimas décadas ha surgido un gran interés por los temas experimentales y conductuales en economía. La magnitud de su importancia se refleja en los Premio Nobel en Economía galardonados recientemente por sus contribuciones en estos campos. En efecto, Vernon Smith y Daniel Kahneman obtuvieron el premio en el año 2002; Richard Thaler en el 2017; Abhijit Banerjee, Esther Duflo y Michael Kremer en el 2019; y David Card, Joshua Angrist y Guido Imbens en el 2021.

Desde la creación del Laboratorio de Economía Experimental (LEEX-PUCP) en 2017, el primero en el Perú, ha habido un creciente interés de los estudiantes por el tema experimental en nuestro país, y en particular en la PUCP. Actualmente, existen (aunque pocos aún) cursos de pregrado y posgrado en esta área.

Este documento tiene como objetivo principal presentar los elementos básicos para programar experimentos en la plataforma oTree. Está dirigido a estudiantes, investigadores y profesionales interesados en elaborar investigación en el área experimental. Es un documento autocontenido y, en principio, no tiene prerrequisitos. No obstante, tener conocimientos previos de teoría de juegos, y de algún lenguaje de programación, facilita la lectura.

El programa oTree es una plataforma de código abierto para programar e implementar experimentos de laboratorio (presencial y en línea) o de campo, o de ambos. Entre las tareas que permite se encuentran las siguientes: encuestas; juegos de estrategias multijugador en general, tales como dilema del prisionero, bienes públicos y subastas; experimentos conductuales controlados en economía, psicología y áreas relacionadas. De acuerdo con su portal, a la fecha, esta plataforma ha sido usada en más 1100 publicaciones académicas.

El presente documento tiene su origen en el material preparado para el Taller Básico de Economía Experimental, organizado por LEEX-PUCP. Este taller se llevó a cabo en julio del 2022, y contó con la participación de 30 estudiantes, y tuvo una duración de 21 horas.

Si bien en la web existe una amplia documentación para programar experimentos en oTree, este es realmente solo un material de consulta. No existe una referencia para aprender lo esencial. Para el dictado del taller mencionado, se tuvo que preparar material, lo cual dio lugar a la primera versión de lo que ahora es este trabajo. Cabe enfatizar que este documento no pretende reemplazar a la documentación, pero sí dar los fundamentos sólidos necesarios para que el estudiante luego profundice. Ciertamente, estas notas tienen como fuente dicha documentación.

Un aspecto importante de estas notas son los ejercicios propuestos que hay en todos los capítulos, excepto el primero por ser más conceptual. El lector, que realmente requiere aprender a programar experimentos en oTree, debe resolver todos los ejercicios en el orden que aparecen, pues se presentan en orden ascendente de grado de dificultad. Para ayudar en esta tarea, en el GitHub de LEEX se dispone de una plantilla, cuando el ejercicio lo requiera. Asimismo, para acceder a los solucionarios de los ejercicios, escriba un correo al laboratorio LEEX-PUCP.

Además de esta introducción, este documento se divide en 4 capítulos, un apéndice y, al final, la bibliografía correspondiente. El capítulo 1 es una introducción breve a la economía experimental. Se presentan los conceptos básicos, los elementos de un experimentos, los tipos de experimentos, y los aspectos éticos en los experimentos. El capítulo concluye con la descripción de experimentos en economía pública, macroeconomía, y finanzas. En el capítulo 2, se presentan los conocimientos mínimos de Python. Esto es importante porque oTree es una estructura basada en Python. Se revisa el manejo de listas y diccionarios, sentencias condicionales y en bucle, así como funciones. También se revisan la creación de clases y métodos. Cabe enfatizar que oTree hace uso intensivo de clases, por lo que se recomienda leer detenidamente este parte. En los capítulos 3 y 4 tratan de la programación de experimentos en oTree. En el capítulo 3 se presentan los elementos básicos, usando como ejemplo la programación de una encuesta predeterminada en oTree. Se describe la instalación de la versión más reciente de oTree, se crea el primer proyecto, se muestra cómo se configura un proyecto, se explora el código y las páginas de un aplicativo. También se detalla cómo ver el experimento en la web y cómo descargar y entender la data luego de ejecutado el experimento. En el capítulo 4 se profundiza en la programación de experimentos, usando como ejemplo el juego de los bienes públicos predeterminado en oTree. El objetivo es presentar los nuevos elementos que aparecen cuando se programa experimentos que implica interacción entre los participantes. En este tipo de experimentos se requiere crear variables a nivel de grupo y alguna función para calcular, entre otras, los payoffs de los jugadores. En ese capítulo también se describen las variables de la data que aparece luego de ejecutar el experimento. Cabe indicar que esta data es compleja, debido a la interacción de los participantes, y porque se ejecuta con más de una ronda. Finalmente, el apéndice presenta los complementos necesarios. Se repasa brevemente algunos comando del CMD de Windows, la instalación y manejo de environments en Anaconda, la instalación de la versión antigua de oTree, así como el manejo de html tags.

El autor y sus colaboradoras agradecen el gran apoyo del Departamento de Economía de la Pontificia Universidad Católica del Perú. Se espera que este trabajo sirva como material de enseñanza en beneficio de estudiantes y docentes del área experimental. Asimismo, que sea la primera guía a todos aquellos investigadores y profesionales interesados en dar sus primeros pasos en la programación de experimentos.

Capítulo 1

Economía experimental

Si bien el objetivo principal de este documento es aprender a programar experimentos, es importante entender los conceptos básicos de la economía experimental. En ese sentido, este primer capítulo es una introducción al tema experimental: se presentan conceptos importantes sobre la economía experimental, su contexto histórico y algunos casos aplicados en donde se ha utilizado esta metodología.

Este capítulo consta de cuatro secciones. En la sección 1 se presentan los conceptos básicos, los elementos de un experimento y los tipos de experimentos. En la sección 2 se discute las condiciones mínimas para que un experimento sea válido, así como los aspectos éticos de los experimentos. En la sección 3 se comenta sobre el software para programar experimentos; y el análisis de datos obtenidos luego de ejecutar el experimento. En la sección 4 se presentan algunos experimentos en economía pública, macroeconomía, y finanzas.

1.1. Conceptos básicos y tipos de experimentos

La economía experimental es una rama de la economía que utiliza diversos métodos experimentales para estudiar el comportamiento del ser humano dentro de un ambiente controlado, ya sea en un laboratorio o en el campo, o estudiando un evento acontecido de forma natural. Esta metodología permite analizar y evaluar teorías microeconómicas, macroeconómicas, políticas públicas, entre otros temas de investigación.

1.1.1. Contexto histórico

Para estudiar brevemente el contexto histórico de la economía experimental, Alvin Roth y John Kagel (1995), en su libro *Handbook of Experimental Economics*, relatan cómo los primeros experimentos en economía empiezan a partir de la década de 1930. Los autores identifican tres corrientes en el siglo XX. La primera de ellas se enfocó en experimentos que analizan las decisiones individuales de los agentes. Entre sus mayores representantes podemos identificar a Thurstone (1931), Wallis and Friedman (1942), Rousses and Hart (1951) y Allais (1953).

De ahí, en la década de los 50, las investigaciones se enfocaron más en experimentos relacionados con la teoría de juegos. Entre los primeros experimentos realizados está el clásico Dilema del Prisionero. Los autores más representativos de esta década fueron Dresher y Flood (1950) y Kalisch, Milnor, Nash y Nering (1954). En el documento realizado por estos últimos, llamado *Some Experimental n-Person Games*, se encuentran varios experimentos clásicos de juegos cooperativos donde la negociación, el regateo y la coalición fueron aspectos claves en la investigación.

En la tercera corriente, las investigaciones estuvieron relacionadas con el tema de la Organización Industrial. Para esto, las investigaciones que más se destacan son las de Chamberlin (1948) y Siegel y Fouraker (1960). El primer autor estudió cómo poner a prueba la teoría neoclásica de la competencia perfecta, mediante distintos experimentos con estudiantes. Además, en años previos, este mismo autor escribió varios trabajos sobre la competencia monopolística, siendo una de las fuentes clásica para el estudio de la Organización Industrial.

En la década del 60 aparece Vernon Smith. Este autor empieza a realizar experimentos dentro de sus investigaciones, inspirado principalmente por Chamberlin, su profesor en Harvard. Entre sus primeros trabajos se destaca el documento *An Experimental Study of Competitive Behaviour*, en el cual evalúa la teoría económica del mercado competitivo (más detalles en la sección 1.1.3). Asimismo, empieza a utilizar los métodos experimentales para evaluar la implementación de políticas públicas en economía, siendo así uno de los primeros dentro del campo (Smith, 1979; Coursey y Smith, 1984). Además de realizar estas investigaciones, escribió documentos sobre cómo aplicar la metodología experimental dentro de la disciplina económica (Smith,1976). En general, Smith ha contribuido de forma extensa a la economía experimental y, por esta razón, parte de sus contribuciones son utilizadas en las distintas secciones de este documento.

Si bien las tres corrientes mencionadas anteriormente fueron el inicio de la economía experimental, a partir de la década de 1970, esta rama tuvo un apogeo mucho mayor. De acuerdo con Friedman y Cassar (2004), hubo un mayor interés de parte de nuevos investigadores, nuevas teorías e innovaciones dentro del laboratorio experimental. Entre los principales investigadores podemos encontrar a Daniel Kahneman y Amos Tversky, desde el área de psicología, y Richard Thaler, desde la economía conductual.

Una de las primeras investigaciones de Daniel Kahneman y Amos Tversky (1974) llamada *Judgment under Uncertainty: Heuristics and Biases*, estudia como los juicios realizados bajo incertidumbre no necesariamente siguen la lógica económica tradicional. Cuando las personas están bajo incertidumbre suelen tomar atajos y ser influenciadas por su heurística, lo cual conlleva a tomar decisiones sesgadas. Asimismo, su otra investigación *Prospect Theory: An Analysis of Decision under Risk*, publicada en 1979, añade que no solo los juicios, sino también la toma de decisiones, se aleja también del comportamiento dictado por la teoría económica clásica.

Richard Thaler realizó grandes contribuciones seminales a la economía y finanzas experimentales. Entre sus primeras investigaciones está *Toward a Positive Theory of Consumer Choice* en el año 1980. En este documento el autor estudia cómo los consumidores, en ciertas condiciones, actúan de manera diferente a cómo lo relata la teoría económica. Se plantea la teoría del *endowment effect*, la cual predice que los agentes van a valorar

más un bien si es que son dueños de este. Después, en la década del 90, Richard Thaler colaboraría junto con Daniel Kahneman para investigar temas relacionados a la aversión al riesgo, la justicia y el *endowment effect*. Su libro "Misbehaving: The Making of Behavioral Economics" resume sus principales contribuciones, y contiene una reseña histórica de la economía conductual, y cómo esta corriente está presente en varios aspectos de la vida cotidiana.

En el año 2002, sucedió un hito importante dentro de la Economía Experimental. Vernon Smith y Daniel Kahneman reciben el Premio Nobel de Economía y esto ayudó a la corriente a ser cada vez más reconocida a nivel mundial. Recientemente, desde la economía experimental se han realizado contribuciones significativas dentro del campo de la economía que también han sido reconocidas por el Premio Nobel. En efecto, Richard Thaler recibió el Premio Nobel de Economía en el año 2017 por sus contribuciones a la economía experimental y del comportamiento. Asimismo, en el 2019, Abhijit Banerjee, Esther Duflo y Michael Kremer fueron galardonados por sus amplios estudios sobre la pobreza y la economía del desarrollo utilizando experimentos de campo para poner a prueba los beneficios de políticas públicas en países con alto índice de pobreza. Del mismo modo, David Card, Joshua Angrist y Guido Imbens, en el año 2021, recibieron el Premio Nobel por sus diversos trabajos sobre economía laboral utilizando experimentos naturales para encontrar relaciones de causa-efecto.

1.1.2. Elementos de un experimento

Para la construcción de un experimento, de laboratorio o de campo, es importante identificar los elementos que lo constituyen. Siguiendo a Vernon Smith (1994) y a Harrison y List (2004), se pueden identificar tres elementos:

a. **Participantes:** son los sujetos dentro del experimento y su comportamiento observado es el input de la investigación. Al conjunto de sujetos dentro de un experimento se le llama "base de participantes" y su naturaleza puede ser estándar y no estándar. En una base de participantes estándar, los sujetos usualmente son estudiantes universitarios. Esto, debido a que es una muestra frecuentemente usada entre los investigadores dentro del campo académico y, de cierta forma, permite que los sujetos no tengan una conexión directa con el tema de investigación. Por otro lado, cuando la base de participantes es no estándar, los sujetos son extraídos del campo de estudio o presentan algunas características relacionadas con él.

b. **Instituciones:** estas aluden a las reglas e instrucciones que permiten el intercambio de información y la correcta realización del experimento. Dentro de las instituciones se determina la naturaleza del bien que se intercambia, las actividades y reglas del experimento.

c. **Ambiente Controlado:** se refiere al entorno en el cual el sujeto se desenvuelve dentro del experimento. Este aspecto se refiere tanto al entorno externo, como a las preferencias, elecciones y dotaciones iniciales de los participantes que serán influenciadas utilizando premios monetarios.

1.1.3. Tipos de experimentos

Según Harrison y List (2004), existen tres tipos de experimentos en Economía: experimentos de laboratorio, de campo, y naturales. Los **experimentos de laboratorio** son aquellos experimentos que se realizan en un laboratorio convencional, con una base de participantes estándar, un ambiente controlado y con reglas establecidas. Un gran representante de esta metodología es el Premio Nobel Vernon Smith. Como ya señalamos, una de sus primeras investigaciones fue *An Experimental Study of Competitive Behaviour*, en la cual el autor lleva a cabo una serie de experimentos dentro de un laboratorio para poner a prueba la teoría neoclásica del mercado competitivo (Smith, 1962).

En este trabajo, Smith dividió a los sujetos de su investigación aleatoriamente en dos grupos: compradores y vendedores. Una vez asignado el rol para cada persona, a los vendedores se les asignó una unidad del bien para que pueda ser vendida y un límite mínimo de precio al cual podían ofrecer esta unidad. A los compradores se les dio, también, un límite de precio del cual no podían sobrepasar para comprar el bien. Una vez establecidas las reglas del juego, los compradores y vendedores podían interactuar entre ellos dentro del mercado creado por el autor. El resultado del experimento es que, con las condiciones pautadas, el precio llegaba a converger al precio de equilibrio en este mercado competitivo simulado.

Por otro parte, los **experimentos de campo** se diferencian de los experimentos de laboratorio en que los primeros utilizan una base de participantes no estándar. A su vez, estos autores dividen este tipo de experimentos en tres categorías:

a. **Experimentos de campo artefactuales**: son aquellos experimentos que se realizan en un laboratorio, con un ambiente controlado, reglas establecidas y con una base de participantes no estándar.

b. **Experimentos de campo contextualizados**: son similares a los experimentos artefactuales, pero se diferencian en el hecho de que el contexto del campo se encuentra presente dentro del experimento, ya sea por la naturaleza del bien que se intercambia, las actividades o la información que el participante pueda utilizar.

c. **Experimentos de campo naturales**: se encuentran en la misma línea que los experimentos de campo contextualizados, con la diferencia de que, en este caso, los participantes actúan de forma natural sin saber que están siendo parte de un experimento.

Específicamente, uno de los trabajos que utiliza experimentos de campo naturales es el realizado por Banerjee et al. (2007) en la India. Este documento estudió los efectos de poder brindar asistencia educacional a los niños de colegios pertenecientes a dos ciudades de la India: Mumbai y Vadodara. Para esto, se eligió de forma aleatoria los colegios que iban a ser beneficiados con el programa con asistentes de docencia. Se concluyó que este tipo de asesorías personalizadas sí ayudaban a mejorar el nivel de educación de los niños en el corto y mediano plazo.

Por último, los **experimentos naturales** son aquellos en los que el experimentador analiza y estudia un acontecimiento histórico en específico. La idea es comparar este acontecimiento con uno o más tratamientos tomando de referencia una línea de base. Una

de las ventajas que proveen los experimentos naturales es que reflejan las decisiones de los individuos en un ambiente real, de manera que, se enfrentan a las consecuencias directas de sus acciones. Por otro lado, entre las desventajas se encuentra el hecho de que el experimentador no elige todas las especificaciones para los grupos de tratamiento, así como tampoco puede elegir el lugar ni el momento en el que ocurre el experimento.

Este tipo de experimentos permitió a David Card y otros encontrar respuestas sobre los efectos que provoca la inmigración, la educación y el salario mínimo en el mercado laboral. En particular, el trabajo de Card y Krueger (1993) analizó si es que un aumento del salario mínimo provocaba una caída en el nivel de empleo. Para ello, examinaron los niveles de empleo en cadenas de *fast food* de dos estados, New Jersey y Pennsylvania. Los autores tomaron como grupo de tratamiento al estado de New Jersey, el cual había tenido lugar un aumento de su nivel de salario mínimo, y lo compararon con el estado de Pennsylvania. Esto permitió que la política pública -aumento del salario mínimo- impuesta en el estado de New Jersey funcionará como un experimento natural. Al final, la conclusión del estudio rompió con los esquemas clásicos dentro del estudio de la economía laboral, ya que se encontró que un aumento del salario mínimo no necesariamente conllevaba a una disminución significativa del nivel de empleo.

1.2. Ideando nuestro primer experimento

En esta sección se va a poner en práctica los conceptos presentados previamente. Se hablará sobre las diferentes propuestas de investigación que existen dentro del campo de la economía experimental, los elementos que se necesitan asegurar para la construcción del experimento y los aspectos éticos a tomar en cuenta.

1.2.1. La propuesta

Según Friedman y Cassar (2004), existen varias propuestas que se pueden realizar para trabajar con un experimento en Economía. Por ejemplo, algunos autores han tratado de evaluar la determinación del precio del mercado; la robustez del equilibrio competitivo; el poder predictivo de las expectativas racionales en equilibrio; entre otros temas diversos. Asimismo, los experimentos no solo son útiles para poner a prueba teorías, sino también para encontrar resultados empíricos en una nueva área de estudio o, para fines más prácticos, como estudiar una política pública para el gobierno o para la industria. Inclusive, la economía experimental es utilizada para entender y predecir las decisiones de compra de los consumidores e influenciar en ellas. Para ver experimentos clásicos en economía, ejecutados por LEEX-PUCP, ver Lopez y Lugon (2020). Asimismo, en la sección 1.4 se describen con cierto detalle experimentos no clàsicos en economía pública, macroeconomía y finanzas.

1.2.2. Construcción del experimento

Entonces, una vez elegido el tema que se quiere, lo siguiente es pensar en cómo será la dinámica del experimento y crearlo. Para esto tenemos que recordar la sección 1.1.2, donde se menciona que existen tres elementos dentro de un experimento: ambiente,

instituciones y participantes. Para asegurar un ambiente controlado, es importante tener en claro la construcción del experimento, cuántos agentes se van a necesitar, cuáles serán sus dotaciones y cómo serán motivados con el incentivo monetario. Por otro lado, para controlar la institución -reglas y normas- del experimento, es necesario que las reglas e instrucciones sean explicadas detalladamente para que los participantes puedan entenderlo de forma clara. Finalmente, para asegurar el tercer elemento, basta con convocar a personas que voluntariamente quieran participar en el experimento, y ofrecerles un incentivo monetario por participar.

A veces puede ser tedioso trabajar con personas como agentes dentro del experimento, porque cada individuo tiene sus propias preferencias y esto puede afectar los resultados. Para ello, se utiliza la teoría del valor inducido presentada por Smith (1976). Esta teoría permite que la recompensa ofrecida a los participantes prevalezca por sobre las preferencia individuales de los agentes y así el experimento no se vea afectado. Para asegurar la prevalencia, Friedman y Cassar (2004) señalan tres condiciones:

- La **monotonicidad** se refiere al preferir tener siempre más. O sea, que los individuos van a tomar una decisión en relación de querer ganar un mayor premio u obtener más del incentivo monetario. Lo que, en general, consiste en siempre querer obtener más dinero, más puntos o trabajar menos para obtener más.
- La **prominencia** consiste en que el premio sea consistente con las acciones realizadas por el agente, o sea, que este sepa que con cierta toma de decisiones puede conseguir la cantidad correspondiente de premio monetario. El sujeto necesita entender completamente el experimento y como sus acciones afectan a su propio resultado.
- La **dominancia** permite que los intereses externos del sujeto, motivos altruistas o de rivalidad, no interfieran por sobre los intereses económicos. Para esto, el anonimato en cada experimento es importante, ya que permite que los sujetos se enfoquen en los premios que pueden conseguir y no en otros aspectos.

1.2.3. Aspectos éticos

Otro aspecto importante dentro de la creación de un experimento son los aspectos éticos. Esto es relevante porque los experimentos que se realicen, así como sus resultados, tendrán un efecto dentro de nuestra sociedad. Esto debido a la aplicación de políticas públicas o de las decisiones económicas que puedan realizar el Estado y las empresas a raíz de estos resultados. Por lo tanto, en la literatura se han propuesto algunos principios claves que se tienen que respetar para realizar un experimento. Según Ifcher y Zarghamee (2015), es necesario respetar tres principios: respeto por las personas, beneficencia y la justicia. Asimismo, además de resaltar estos tres principios, el Comité de Ética para la Investigación PUCP añaden el de integridad científica y responsabilidad. De esta forma, estos serían los cinco principios que hay que seguir para llevar a cabo un experimento:

1. **Respeto por las personas**: dentro de este principio es importante tomar en cuenta que el ser humano no es un objeto ni un instrumento dentro del estudio; por lo tanto, se debe respetar su completa autonomía dentro de la investigación.

2. **Beneficencia y no malencia**: para este principio es importante el bienestar del participante, el experimentador se tiene que asegurar que el participante no se vea afectado negativamente o reciba algún daño, ya sea físico o mental.
3. **Justicia**: el experimentador aquí tiene la obligación de tratar con igualdad y equidad a los sujetos que participan en el experimento.
4. **Integridad Científica**: se tiene que tener un cuidado especial con los datos e información que el experimentador obtenga de los participantes. Además, se tiene que prevenir los conflictos de intereses entre el experimentador y los participantes.
5. **Responsabilidad**: dentro de este principio se resalta la conciencia personal el experimentador, de forma que, las decisiones que este realice dentro de su investigación tendrán consecuencias que quizás no puedan ser observadas.

Cabe indicar que todos los experimentos llevados a cabo en LEEX - PUCP cuentan con la aprobación del Comité de Ética para la Investigación de la Universidad.

1.3. Ejecución del experimento y análisis de datos

1.3.1. Sotfware para experimentos

Una vez elegido el tema de investigación y considerados los aspectos centrales en la construcción del experimento, un aspecto indispensable es la elección del software para llevar a cabo el experimento. Podemos encontrar varios software para programar experimentos en el mundo académico, tales como oTree y Z-tree. En este documento, explicamos los fundamentos de programación de un experimento en oTree en los capítulos 3 y 4, el cual es el software que se usa actualmente en el Laboratorio de Economía Experimental LEEX-PUCP.

OTree es una plataforma de código abierto que permite realizar tareas interactivas basadas en la web. El trabajo de Otree se encuentra enlazado con el lenguaje de programación Python, por esta razón en el capítulo 2 se presentan los conceptos básicos para aprender este lenguaje, tales como la creación de listas, bucles y diccionarios. Entre los experimentos predeterminados que se pueden realizar en esta plataforma se tiene el juego de bienes públicos, Cournot, dilema de prisionero, subastas, encuestas, cuestionarios, entre otros.

1.3.2. Análisis de datos

Finalizado el experimento, lo que sigue es analizar los datos obtenidos. Para lo cual se recomienda realizar dos fases de análisis: cualitativa y cuantitativa. Para la **fase cualitativa**, uno se tiene que familiarizar con los datos obtenidos y empezar a preguntarse por la lógica de los resultados. Para esto, es importante apoyarse en la teoría para reflexionar acerca de lo que esta predice sobre las relaciones de las variables de estudio.

Por otro lado, para la **fase cuantitativa** es necesario trabajar con los datos directamente y, para realizar esto, se recomienda seguir dos etapas. La primera es la **estadística des-**

criptiva, en la cual el experimentador tiene que elaborar gráficos y tablas estadísticas para describir las características de sus datos (Jacquement y L'Haridon, 2018). Entre los gráficos más utilizados se encuentran los histogramas, gráficos de caja y gráficos de dispersión. Del mismo modo, la información que usualmente se encuentra dentro de las tablas estadísticas es la del promedio y la mediana de las variables.

Terminado la estadística descriptiva, la siguiente etapa es la del **análisis inferencial**. En esta etapa se necesita comparar las respuestas promedios de los participantes entre los grupos de tratamiento que se disponga (Friedman y Cassar, 2004). Además, se tienen que evaluar las diferencias que se obtengan con las técnicas estadísticas correspondientes al modelo que se quiera estudiar. Dentro de esto, es indispensable estimar coeficientes significativos que ayuden a explicar las relaciones entre las variables (Jacquement y L'Haridon, 2018). Para detalles de técnicas estadísticas y econométricas para datos experimentales, ver, por ejemplo, Moffat (2015) y Gerber y Green (2012).

1.4. Experimentos en economía

En 2017, año de la fundación del Laboratorio de Economía Experimental, LEEX-PUCP, se llevó a cabo los siguientes experimentos clásicos: el juego del ultimátum, el juego de bienes públicos, el concurso de belleza keynesiana y el juego de confianza. Los detalles se encuentran en Lopez y Lugon (2020).

En esta última sección del capítulo, se van a presentar con cierto detalle algunos ejemplos de experimentos publicados en la literatura. Para mostrar la amplitud de las aplicaciones, se presentan experimentos dentro de la Economía Pública, Macroeconomía y Finanzas. Otros experimentos se presentan en Brañas (2011), y Galarza (2012).

1.4.1. Experimentos en economía pública

En el área de Economía Pública, un tema de estudio recurrente es la asignación de bienes públicos y el comportamiento del free rider. El juego de bienes públicos tiene como propósito analizar este comportamiento mediante el mecanismo de contribución voluntaria (Isaac et al., 1984). Aquí encontramos estudios experimentales como el de los sociólogos Marwell y Ames (1979), el cual es considerado el primer estudio experimental de bienes públicos. Posteriormente, se realizaron otros experimentos como el de Kim y Walker (1984), y el de Isaac y Walker (1988). El primero hace referencia al problema del free rider, y el segundo analiza el efecto del tamaño del grupo en las contribuciones voluntarias. Por otro lado, también se ha estudiado el comportamiento de los individuos ante una intervención de una figura de autoridad. Aquí tenemos estudios como el de Barron y Nurminen (2020), el cual se explicará a continuación.

Este artículo es uno de los más recientes respecto al juego de bienes públicos. Su investigación consiste en dos intervenciones de una figura de autoridad para promover la cooperación cuando se desconoce el valor preciso de las contribuciones de los individuos. Estas intervenciones son: 1) implementar un nudge, que llamaremos intervención nudge y; 2) obligar a los participantes a revelar su contribución ex post anónimamente, que llamaremos intervención revelación. La primera intervención consiste

en aplicar un "empujón" (nudge) para obtener una contribución más alta de los participantes. Esto se logra etiquetando las contribuciones, es decir, las que pasen cierto umbral, 80 % de la dotación en este caso, serán catalogadas como "buenas". Así, los individuos tendrán un punto focal claro al momento de considerar la cantidad a aportar. Por otro lado, la segunda intervención explora la aversión a la mentira mediante la exigencia a los individuos de revelar su contribución ex post de manera anónima. De esta manera, se busca inducir una mayor contribución, ya que se supone que los individuos experimentan desutilidad por anunciar una contribución baja. El aporte del estudio es evidenciar que, en contextos de dilemas sociales, en los que es difícil observar las acciones de los individuos y fomentar la cooperación, la provisión de un empujón transparente y de conocimiento público puede ser efectivo para conseguir una mayor contribución.

En el diseño del experimento, los grupos de interés son el grupo de control, el grupo que fue sometido solamente a la intervención nudge, y el grupo que fue sometido solamente a la intervención revelación. En cada uno de ellos, los participantes forman grupos de cuatro de manera aleatoria y cada uno posee una dotación de veinte puntos, la cual se divide entre la contribución para el proyecto grupal y la cantidad con la que decide quedarse. Si el sujeto se queda con un punto, sus ingresos aumentan en un punto. En cambio, si decide aportar al grupo, se suman todos los puntos, se multiplica por 1,4 y se divide de manera equitativa entre los cuatro miembros del grupo. El pago final del individuo i se define con la siguiente ecuación:

$$\pi_i = 20 - g_i + \frac{1{,}4}{4} \sum_{j=1}^{4} g_j \, ; \quad i = 1, 2, 3, 4 \tag{1.1}$$

Donde g_j denota la contribución del miembro j al grupo.

Los resultados del experimento muestran que si se aplica un empujón, hay una mayor contribución. Exactamente, se observa que los sujetos contribuyen en promedio 40 % más en comparación al grupo de control. Asimismo, la contribución mediana coincide con el valor del punto focal (80 % de la dotación); mientras que, en los otros dos grupos sin el empujón (grupo control y grupo revelación), se observa que la contribución mediana es menos del 40 % de la dotación. En particular, en el grupo revelación, se evidencia que no existe una influencia en el aumento del nivel de las contribuciones respecto al grupo control, y que el 27 % de los que aportaron cero al bien público mintieron sobre su nivel de contribución en su revelación anónima. Una posible explicación de este comportamiento es que el costo de anunciar su elección egoísta es mayor al costo de decir una mentira anónima.

1.4.2. Experimentos en macroeconomía

En la macroeconomía moderna, un área de estudio relevante es la formación de expectativas de los agentes económicos. En este campo, los primeros trabajos experimentales consistieron en que los participantes pronosticaran la trayectoria del precio de un activo presentándoles el proceso generador de datos (Schmalensee, 1976; Smith, Suchanek y Williams, 1988). Sin embargo, la mayoría de estos primeros experimentos no permitían la retroalimentación; es decir, las expectativas de resultados futuros de

los agentes no eran consideradas en la determinación de los resultados presentes. A partir de Marimon y Sunder (1993, 1994, 1995), esta literatura empezó a incorporar la retroalimentación en los resultados experimentales.

En la mayor parte de las investigaciones, el modelo estándar es de expectativas racionales (ER). Para estudiar si los sujetos pueden aprender un equilibrio de ER se han desarrollado dos enfoques: experimentos de aprendizaje para pronosticar (*learning-to-forecast experiment:* LtFE) y experimentos de aprendizaje para optimizar (*learning-to-optimize experiment:* LtOE). Por un lado, en el enfoque experimental LtFE se pide a los participantes que presenten un pronóstico de alguna variable económica futura y se les recompensa en base a la precisión de su pronóstico. Dentro de las investigaciones que utilizan este enfoque se encuentra Hommes et al. (2007), Adam (2007), entre otros. Por otro lado, en el enfoque experimental LtOE se les pide a los participantes que tomen decisiones económicas, como producir o consumir, directamente, sin obtener de ellos ningún pronostico. Las cantidades de consumo o producción que elijan los agentes determinan implícitamente la realización de precios que se buscan pronosticar. Entre las investigaciones que utilizan este enfoque encontramos a Smith et al. (1988), Noussair et al. (2007), entre otros.

Un trabajo pionero que compara y combina un diseño LtFE con un LtOE es el de Bao et al. (2013). Estudian cómo mejora o empeora la convergencia hacia el Equilibrio de Expectativas Racionales (EER) y la velocidad de dicha convergencia. El diseño del experimento consta principalmente de 3 elementos: el marco teórico, los tratamientos y sus respectivos esquemas de pago. En primer lugar, el experimento se desarrolla sobre la base de un modelo simple de economía de telaraña de N empresas, la cual es un sistema de retroalimentación negativa; es decir, un precio esperado más alto conduce a una mayor producción, y, por ende, un precio de mercado actual bajo. Este modelo es un mercado de oferta y demanda de un solo bien. La demanda es una función lineal decreciente en los precios, y está dada exógenamente. Mientras que la oferta total, bajo el supuesto que las empresas tienen la misma función de oferta, es la suma de todas las ofertas individuales que se obtienen a partir de la maximización del beneficio esperado en un periodo t. Los participantes juegan el rol de empresas en el experimento. Asimismo, dentro del modelo teórico, se impone el supuesto de ER, el cual es que en promedio las expectativas de precios convergen a un precio de equilibrio, siendo este $p^* = 30.73$. Todo lo anterior podemos visualizarlo en las siguientes ecuaciones:

Sea la siguiente función de demanda de mercado:

$$D\left(p_t\right) = 63 - \frac{21}{20} p_t \tag{1.2}$$

Se asume que cada empresa tiene una función de costos cuadrática, la cual es:

$$c\left(q_{h,t}\right) = \frac{H q_{h,t}^2}{2}, \text{ donde H es el total de empresas} \tag{1.3}$$

Utilizando la ecuación (1.3), el beneficio esperado por una empresa h está dada por:

$$\pi_{h,t}^e = p_{h,t}^e q_{h,t} - c\left(q_{h,t}\right)$$

Resolviendo el problema de maximización de beneficios obtenemos que la oferta óptima de cada empresa h es:

$$S^*\left(p^e_{h,t}\right) = \frac{p^e_{h,t}}{H}$$

Obteniendo la oferta agregada, vemos que esta es igual al promedio del precio esperado:

$$\sum_h S^*\left(p^e_{h,t}\right) = \overline{p^e_t} \tag{1.4}$$

Sustituyendo la ecuación (1.4) en la ecuación de la demanda (1.2), obtenemos lo siguiente:

$$p_t = max\left\{\frac{20}{21}\left(63 - \overline{p^e_t}\right), 0\right\}$$

Agregando un *shock* al precio de equilibrio ϵ_t, de variables aleatorias i.i.d, con distribución normal estándar, se tiene:

$$p_t = max\left\{\frac{20}{21}\left(63 - \overline{p^e_t}\right) + \epsilon_t, 0\right\}$$

Por último, imponiendo el supuesto de ER, $p^* = E[\frac{20}{21}(63 - p^*) + \epsilon_t]$, obtenemos el precio de equilibrio, el cual es $p^* = 30.73$, la oferta óptima es 5.12 y el beneficio óptimo para cada empresa es 78.70.

En segundo lugar, se presentarán los siguientes dos elementos principales del experimento de Bao et al. (2013): los tratamientos y los esquemas de pago. Al final, también mencionaremos algunos detalles importantes sobre el desarrollo del experimento. Con respecto al número de tratamientos, el estudio analiza cinco grupos de tratamientos, de los cuales solo se describirán los cuatro principales. Los tratamientos difieren en las tareas asignadas a los participantes, y en el esquema de pagos. Con respecto a este último está diseñado para que en todos los tratamientos se incentive a los agentes a reducir sus errores de predicción, generando que en cada periodo sean más precisos. A continuación, se presentarán los tratamientos con sus respectivos esquemas de pago.

En el primer tratamiento, los sujetos solo realizan pronósticos de precios (LtFE). Dado el pronóstico, se calcula la decisión de oferta óptima de cada individuo. A continuación, se halla la oferta total y se iguala con la demanda agregada exógena, obteniendo así el precio de mercado. Luego, a cada sujeto se le paga de acuerdo con la diferencia entre su precio de su pronóstico $p^e_{h,t}$ y el precio del mercado real p_t. Específicamente, la función de pagos para el individuo h es:

$$max\left\{1300 - \frac{1300}{49}\left(p_t - p^e_{h,t}\right)^2, 0\right\}$$

En el segundo tratamiento, los sujetos solo toman decisiones sobre la cantidad ofertada (LtOE). El precio de mercado está determinado por las decisiones de producción de

todas las empresas. Luego, a cada sujeto se le paga de acuerdo al beneficio que su empresa obtiene en cada periodo:.

$$p_t q_{h,t} - 3\left(q_{h,t}\right)^2 + 1200$$

En el tercer tratamiento, cada sujeto desempeña dos roles: pronosticador y director de producción. En este caso, el precio de mercado se va determinar como en el caso del tratamiento dos. Luego, el pago que recibe cada sujeto es una combinación lineal de igual ponderación de las funciones de pago de los tratamientos anteriores.

En el cuarto tratamiento hay un equipo de dos participantes que representan juntos a una empresa. Uno se encarga de pronosticar, y con esa predicción, el otro decide la cantidad de producción. El precio de mercado se halla con la condición de la oferta total igual a la demanda total exógena, siendo la oferta total la suma de todas las decisiones de producción de las empresas en un mercado. A cada sujeto se le paga como en el tercer tratamiento.

Respecto al desarrollo del experimento es importante mencionar lo siguiente. En todos los tratamientos, cada sujeto puede leer sus pagos potenciales para la tarea de pronóstico o para la tarea de producción (optimización). Asimismo, cada tratamiento es realizado en sesiones diferentes, por lo que todos los sujetos en una misma sesión enfrentaron la misma condición de tratamiento. En total, se realizan nueve sesiones; en cada sesión se agruparon a los participantes en grupos de 6 (12 solo para el tercer tratamiento) y hubo entre 4 a 7 grupos por sesión dependiendo del tratamiento. Para el experimento, cada grupo representa un mercado en el que se define el precio de mercado dadas las decisiones de las empresas en cada ronda. En total se realizaron 50 rondas por sesión.

Los resultados del experimento muestran que el precio promedio de todos los tratamientos converge al precio de EER, pero a distintas velocidades. Asimismo, se observa que ese hallazgo no se ve afectado por el tipo de tarea (pronosticar u optimizar) que se asigna al participante. Sin embargo, la velocidad de convergencia depende de la tarea asignada: en la tarea de pronosticar los sujetos parecen aprender más rápido el precio de ERR que bajo la tarea de optimizar. Adicionalmente, se observa que el ajuste más rápido al equilibrio es del primer tratamiento; mientras que el más lento es en el tercer tratamiento. Por último, el cuarto tratamiento es mejor que el tercero en cuanto a la rapidez de encontrar el precio de EER, lo que sugiere que en una empresa debería haber especialización.

1.4.3. Experimentos en finanzas

En el área de Finanzas se han realizado diversas investigaciones para entender el comportamiento de los agentes y sus decisiones. Los temas más recurrentes que se pueden encontrar dentro de esta rama son los estudios sobre las decisiones bajo riesgo e incertidumbre, gestión del portafolio, creación de burbujas especulativas y las decisiones intertemporales de ahorro y gasto.

Entre los autores más reconocidos en estos temas está Richard Thaler, autor previamente mencionado en la sección 1.1.1. Una de sus investigaciones más destacas en el área

de finanzas es *The Effect of Myopia and Loss Aversion on Risk Taking: An Experimental Test* (Thaler et al., 1997). En este trabajo, los autores utilizan métodos experimentales para estudiar el efecto de las decisiones de los inversores cuando presentan una aversión miope a la pérdida. Por otra parte, también, podemos identificar otros autores como Shefrin y Statman (2000) y Moinas y Pouget (2013) que también han colaborado con las finanzas experimentales estudiando temas de portafolio y burbujas especulativas, respectivamente.

A continuación describimos la investigación de Moinas y Pouget (2013). Este es un estudio experimental que trata de explicar cómo surge la creación de burbujas especulativas dentro del mercado financiero. Para esto, los autores proponen un *bubble game* donde se presenta un mercado secuencial para un activo que no tiene valor. Ninguno de los participantes dentro del juego está seguro de ser el último que propone un precio en el mercado y no pueden ver las acciones pasadas de los otros participantes. El juego, por lo tanto, va a tratar de estudiar si es que bajo estas condiciones se crean burbujas especulativas cuando no hay un límite de precio.

Acorde con la teoría que siguen los autores, el efecto cuando hay un límite de precio es importante dentro del experimento por dos razones. Primero, porque cuando hay un límite de precio, o sea, la economía tiene un monto limitado de riqueza, las burbujas que se forman son irracionales. Esto debido a que cuando el participante se da cuenta que es el último en el mercado se rehúsa a comprar el activo y si, incluso, es el penúltimo también se rehúsa porque anticipa que el último no lo comprará (*inducción hacia atrás*). Por otro lado, cuando no hay límite de precio, las burbujas son racionales debido a que ningún participante sabe si es que es el último dentro del mercado, así que hay mayor predisposición a seguir comprando. Por esta razón, los autores arman un experimento donde se estudie esta racionalidad de *inducción hacia atrás* para ver si la formación de burbujas depende de esto o no.

Para armar el diseño del experimento los autores se aseguran de distintos elementos. Primero, se crea un mercado secuencial para la transacción de un activo con valor 0 entre tres participantes. Segundo, se organizan cuatro grupos de tratamiento que dependen de la existencia y del nivel del límite del precio. El límite es representado por la letra K en un intervalo de precio $1 < P < 100K$, donde K toma los valores de 1, 100, 10,000 e ∞. El valor ∞ significa que el precio no tiene límite.

A cada uno de los participantes se le asigna una posición en la secuencia del mercado, primero, segundo y tercero con la misma probabilidad de 1/3. Sin embargo, para evitar que cada participante conozca su posición en el mercado, ellos juegan al mismo tiempo. Aun cuando ningún participante conoce verdaderamente su posición dentro del mercado, pueden inferir en qué posición están según la información que obtengan al ver el precio del activo que quieren comprar. Los precios se generan aleatoriamente y son potencias de 10. Cada participante tiene una dotación de una unidad de capital y, además, puede tener capital adicional – para comprar el activo – provisto por un financiero externo. En este caso, el financiero externo es el experimentador. Tanto el financiero externo y el participante consiguen una ganancia cuando logran vender el activo, este último obtiene 10 unidades de capital cuando vende el activo.

Por lo tanto, la dinámica del juego consiste principalmente en que cada jugador va a

tener la posibilidad de comprar o no comprar el activo al precio que se le presenta. Como se mencionó previamente, cada jugador tendrá una dotación de 1 unidad de capital y si el precio al que se quiere comprar excede de su dotación, el jugador puede comprar el activo sin restricciones, debido a que lo restante será provisto por el financiador externo.

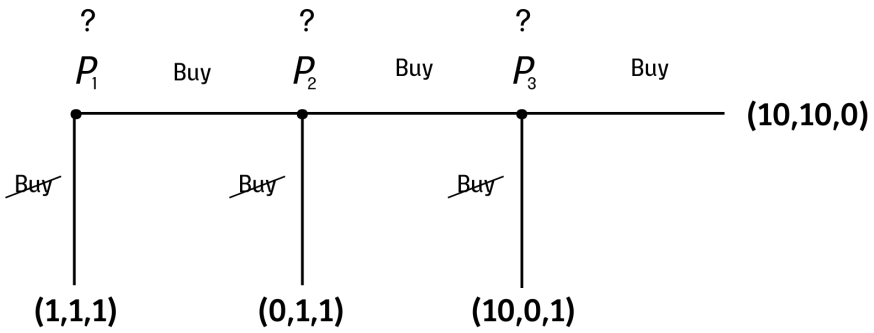
Tabla 1: Tabla de Probabilidades para The Bubble Game

Precio Inicial	Probabilidad de que este precio inicial aparezca
1	50 %
10	25 %
100	12.5 %
1,000	6.25 %
10,000	3.13 %
100,000	1.56 %

Fuente: Huber (2018)

El precio presentado al primer jugador es seleccionado aleatoriamente y, de este precio, va a depender los del resto. La probabilidad de que aparezca cada precio inicial se encuentra en la tabla de arriba. Entonces, si al primer jugador se le presenta el precio $P1$, este tendrá la posibilidad de comprar el activo a ese precio y de ahí venderlo al segundo jugador a un precio 10 veces mayor, el cual sería $P2 = P1x10$. Si el primer jugador vende el activo al segundo jugador con el precio $P2$, este último tiene la posibilidad de venderlo al tercer jugador a un precio 10 veces mayor, en este caso sería $P3 = P2x10$. Si el tercer jugador compra el activo al precio $P3$, el segundo jugador obtiene ganancias. Sin embargo, el tercer jugador no podrá revender el activo, puesto que es el último del turno. Por lo tanto, en este caso, el tercer jugador no tendría ganancias, mientras que los otros dos jugadores sí. Para resumir este intercambio se presenta la imagen a continuación:

Figura 1: The Bubble Game



Fuente: Moinas y Pouget (2013)

El gráfico explica la dinámica del juego para cada participante. Si el primer participante no compra el activo, entonces el mercado secuencial no continúa y todos los participantes se quedan con sus dotaciones, el escenario plantea que las ganancias sean iguales

$(1, 1, 1)$. Por otro lado, si compra el activo, entonces tiene la posibilidad de vender ese activo al participante 2 y obtener una ganancia de 10 unidades de capital. Si el participante 2 decide no comprar el activo, entonces, el participante 1 no obtiene ganancia y se queda con 0, mientras que los participantes restantes se quedan con sus dotaciones $(0, 1, 1)$. En cambio, si el participante 2 compra el activo, el participante 1 obtiene 10 de ganancia. El activo, ahora en posesión del participante 2, puede ser vendido al participante 3, pero si este último decide no comprar, las ganancias solo las habrá recibido el primer participante y el segundo quedará en cero $(10, 0, 1)$. Pero, si el participante 3 compra el activo, entonces el participante 2 obtendrá una ganancia de 10, al igual que el participante 1. El único participante que se queda sin ganancias en este caso es el tercero, porque al ser el último de los participantes, no tiene a quién venderle $(10, 10, 0)$.

Al final los resultados principales del experimento fueron dos. El primero de ellos fue de que las burbujas se formaban, ya sea, exista un límite de precios o no. Incluso se llegaban a formar cuando estaba presente la racionalidad de *inducción hacia atrás*. Por otro lado, el otro resultado fue que la predisposición de un sujeto en entrar a una burbuja incrementa con la distancia entre el precio ofrecido y el precio máximo, *snowball effect*.

Capítulo 2

Python: El Lenguaje de Programación Básico

Como se mencionó en el Capítulo 1, oTree es una de las plataformas de código abierto más poderosas para el desarrollo de experimentos en ciencias sociales y la investigación del comportamiento. Siendo oTree una estructura basada sobre un formato de código Python, en este capítulo presentamos una guía básica para aprender este lenguaje de programación. Para el desarrollo de este capítulo cualquier versión de Python es útil. Cabe señalar que Python permite escribir programas compactos y legibles, los cuales son típicamente más cortos que en C, C++ o Java.

Es importante mencionar que para el desarrollo de este capítulo se utilizó la aplicación Jupyter Notebook de Anaconda[1], la cual permite escribir códigos de manera amigable. Para acceder a Jupyter Notebook, escriba en el buscador de Windows "Jupyter" y haga click. Allí se abrirá una ventana en el navegador y podrá crear una carpeta que puede nombrar "Tutorial_Python_Leex". Luego, dentro de esa carpeta, cree un archivo de python que puede nombrar *tutorial.ipynb* y podrá ejecutar los códigos que verá en este capítulo[2].

La revisión de Python de este capítulo tiene diez secciones. La sección 1 define algunos términos geneales como objeto, método y variable. La sección 2 trata de los objetos más simples, tales como números, cadenas de texto y tipos booleanos. La sección 3 revisa los operadores lógicos y matemáticos. La sección 4 describe las operaciones y métodos con cadenas de texto. La sección 5 presenta el tema de listas y sus principales métodos. La sección 6 aborda el tema de la creación y operaciones con diccionarios. La sección 7 trata sobre sentencias condicionales y en bucles. La sección 8 estudia las sintaxis de las funciones; en particular, la elaboración de funciones. La sección 9 presenta una introducción al tema de clases y métodos; con énfasis en la creación de clases. Finalmente, la sección 10 contiene ejercicios propuestos.

[1]Para la instalación de Anaconda, ver Apéndice A.2.
[2]Para mayor información, consulte la Documentación de Python

2.1. Conceptos generales

En esta primera parte se mencionan algunos términos que se utilizarán a lo largo de todo el capítulo.

¿Qué es un objeto?

Todo lo que puede crear es un objeto en Python. Todo objeto tiene un tipo. Por ejemplo, existen tipos de objetos simples porque son unidades como: int (entero), float (real), str (cadena de texto), boolean (valores True/False), etc. Por otro lado, existen otros objetos que pueden tener una cantidad arbitraria de datos de cualquier tipo, por lo que pueden ser de tipo lista, tupla, diccionario, función, etc.

Asimismo, existen objetos mutables e inmutables. Los objetos mutables son aquellos que pueden cambiarse su contenido, como las listas. Mientras los objetos inmutables, como las tuplas, no pueden ser modificadas.

El Cuadro 2.1 hace referencia a algunas de las categorías en las que los tipos de objetos pueden dividirse.

Tipo	**Nombre**
Texto	cadena de texto (str)
Números	interos(int), flotantes(float)
Compuestos	lista(list), tupla(tuple), rango(range)
Mapeo	diccionarios(dict)
Conjuntos	conjuntos(set), frozenset
Booleanos	bool
Binarios	bytes, bytearray, memoryview

¿Qué es un método?

El método es una función o tarea que pertenece a un objeto. La sintaxis usual para utilizar un método es **obj.metodo**, donde *obj* es el nombre del objeto y *metodo* es el nombre del método que está definido por el tipo de objeto. Es decir, distintos tipos de objetos tienen distintos métodos. Por ejemplo, Python ofrece una serie de métodos ya creados, como los del objeto tipo lista: append, insert, remove, sort, etc; o como los del objeto tipo cadena de texto (*string*): capitalize, upper, long, title, etc. Más adelante, verá ejemplos de los métodos mencionados de estos objetos.

Asimismo, aunque los métodos se llamen igual para distintos tipos de objetos, tienen diferentes significados. Por ejemplo, count es un método tanto de los *strings* como de los objetos *list*. En el primero cuenta cuántas veces se repite una letra en una cadena de texto y en el segundo cuenta las veces que se repite un elemento en una lista.

- `count de lista`

[83]:
```
lst = [1, 1, 2, 3, 1, 5, 3, 3, 1, 3, 1, 8]
lst.count(1)
```

[83]: 5

- count de string

[1]:
```
x = "experimento"
x.count("e")
```

[1]: 3

Por otro lado, también es posible crear sus propios objetos y métodos usando **Clases**, tema que será desarrollado en la sección 2.9.

Tener en cuenta que existen funciones en Python predefinidas que no son métodos. La principal diferencia entre un método y función es que el primero es parte de una clase (está asociado a un objeto), mientras la segunda no necesita pertenecer a ninguna clase.

Veamos un ejemplo de una función que no es un método: `print()`. Esta función puede imprimir uno o más objetos simultáneamente. Puede contener números, cadenas de texto, o variables ya creadas junto con expresiones. Vea los siguientes ejemplos.

Dentro de la función `print()` puede usar cadenas de texto y expresiones numéricas.

[8]:
```
print("Nuestro promedio de notas fue")
print((13 + 17 + 15)/3)
```

```
Nuestro promedio de notas fue:
15.0
```

También, se pueden llamar variables (concepto que luego se desarrollará) dentro de `print()`. En el siguiente ejemplo se crea la variable *b* que representa una cadena de texto, y la variable *a* que representa un flotante; al mismo tiempo, se define un objeto dentro de la función, como un *string*. Dentro de la función `print()` se colocan las variables creadas y el objeto *string* separados por comas.

[10]:
```
b = "Nuestro promedio de notas fue"
a = (13 + 17 + 15)/3
print(b, a, "en el curso de Economía")
```

```
Nuestro promedio de notas fue 15.0 en el curso de Economía
```

Otra forma sencilla de introducir variables y cadenas de texto es con la siguiente notación: *f"...{var} ... {var}"*.

[11]:
```
b = "Nuestro promedio de notas fue"
a = (13 + 17 + 15)/3
print(f"{b} {a} en el curso de Economía")
```

```
Nuestro promedio de notas fue 15.0 en el curso de Economía
```

¿Qué es una variable?

Ya se han creado variables sin saber su definición. Las variables son "etiquetas" asociadas a un objeto. Es un identificador o nombre asignado a un objeto. Para tener más claro

lo anterior vamos a utilizar el operador 'is' para determinar si dos variables (identificadores) hacen referencia al mismo objeto. Se utiliza el operador igual (=) para asignarle el valor a la variable.

Primero, las etiquetas a y b se asignan a un objeto de tipo int (entero). Para un objeto de este tipo asignarle la misma etiqueta a un mismo valor del objeto es suficiente para observar que *a* es igual a *b*. **Notar que** para escribir comentarios dentro de un bloque de código en Python usamos el signo #.

[3]:
```
a = 1
# El objeto 1 (que es de tipo número entero 'int') ahora se llama 'a'
b = 1
# 'b' es una segunda etiqueta asociada al objeto 1
a is b
# La variable a es igual a la variable b
```

[3]:
```
True
```

Observación:

- Las etiquetas 'a' y 'b' han sido asignadas para diferentes objetos. Al llamar a esas variables puede ver a los últimos objetos a los que etiquetaron. Es decir, ahora mismo, en Python, $a = 1 y b = 1$. No puede acceder a los objetos anteriores a través de esas mismas etiquetas, ya que son 'olvidados' inmediatamente.

2.2. Tipos de objetos simples

A continuación, se desarrollarán 4 tipos de objetos simples de Python que ya ha estado utilizando anteriormete: entero (int), flotante (float), cadena (string) y booleano (boolean).

Tipo Numéricos (integers and floats)

Entre los objetos numéricos puede encontrar los enteros y flotantes (decimales). Observe los siguientes ejemplos, en los que se identifica el tipo o clase de objeto con el comando type(). Es decir, la función anterior sirve para determinar el tipo de variable.

Entero o *integer*

[2]:
```
x = 3
print(f'{x} es {type(x)}')
```

```
3 es <class 'int'>
```

Decimal o *float*

[2]:
```
# Tipo float
y = 12.234
print(f'{y} es {type(y)}')
```

```
12.234 es <class 'float'>
```

Tipo texto

Como ya ha visto, la función print() produce una salida más legible de una cadena de texto en comparación si la llamamos simplemente por su etiqueta. Observe el siguiente ejemplo.

[6]:
```
s = "Hola mundo"
print(s)
s
```

```
Hola mundo
```

[6]:
```
'Hola mundo'
```

Ahora, se presentan diferentes formas de crear cadenas de texto.

Primero, se crea una cadena de texto usando comillas dobles y otra con comillas simples.

Comillas dobles

[2]:
```
comment1 = "Me gustan los experimentos
print(f'{comment1} es {type(comment1)}')
```

```
Me gustan los experimentos es <class 'str'>
```

Comillas simples

[2]:
```
comment2 = 'Me gusta la economía
print(f'{comment2} es {type(comment2)}')
```

```
Me gusta la economía es <class 'str'>
```

También, puede utilizar ambos tipos de comillas para que al usar la función print() se visualicen las comillas.

[2]:
```
comment3 = "'Soy economista'"
print(f'{comment3} es {type(comment3)}')
```

```
'Soy economista' es <class 'str'>
```

Asimismo, puede utilizar tres comillas dobles o simples para escribir un texto en varias líneas.

[2]:
```
mensaje1 = """
Tres comillas dobles para
escribir un comentario
en varias líneas
"""
print(mensaje1)
```

```
Tres comillas dobles para
escribir un comentario
```

```
en varias líneas
```

[2]:
```
mensaje2 = '''
Aquí utilizo tres
comillas simples
'''
print(mensaje2)
```

```
Aquí utilizo tres
comillas simples
```

Tipo booleanos

Los booleanos son tipos de variables que indican verdadero (True) o falso (False), siendo su codificación 1 y 0, respectivamente.

[3]:
```
c = True
d = False
print(f'{c} y {d} son {type(c)}')
print(int(c), int(d)) # para que te devuelva el valor numérico de la␣
 ↪variable categórica, se utiliza "int()"
```

```
True y False son <class 'bool'>
1 0
```

2.3. Operadores matemáticos y lógicos

Operadores aritméticos

Python puede utilizarse como una calculadora. Entre los principales operadores encontramos: (+) suma, (-) resta, (*) multiplicación, (/) división y (**) potencia. Asimismo, para obtener la parte entera de una división puede utilizarse el operador (//), mientras para calcular el residuo de la división se utiliza el operador (%). Los paréntesis pueden ser usados para agrupar. Observe los siguientes ejemplos.

[6]:
```
print(2 + 4)
print(120 - 4 * 5)
print((120 - 4) * 5)
print(120 - 4 * 5/4)# Una división siempre devuelve un número flotante
print(17 / 3)
print(17 // 3)
print(17 % 3)
print(2 ** 7)
```

```
6
100
580
115.0
5.666666666666667
```

```
5
2
128
```

Recordar que con el signo '=' puede asignar valor a una variable. Es posible hacer operaciones matemáticas con las variables creadas. Observe el siguiente ejemplo.

[7]:
```
ancho = 20
largo = 10
área = ancho * largo
área
```

[7]:
```
200
```

Por otro lado, podemos usar guión abajo '_' para hacer referencia a la última expresión impresa. Veamos el siguiente ejemplo, en donde la última expresión fue el número entero 200 llamado *área*.

[9]:
```
áreax3 = _ * 3 # multiplicamos el área por 3
áreax3
```

[9]:
```
600
```

Operadores de comparación

La comparación entre objetos o variables se realiza con los siguientes operadores estándar:

- > Mayor que
- < Menor que
- >= Mayor o igual a
- <= Menor o igual a
- == Igual a
- != Distinto a

El resultado de la comparación genera un objeto booleano: True o False, como podemos observar en los siguientes ejemplos.

[9]:
```
print(5 < 4)
print(3.14 >= 3.141)
print(4 != 4.23)
print(10 == 10)
```

```
False
False
True
True
```

Operadores lógicos

Los siguientes operadores evalúan dos expresiones generando un objeto True o False

- `and`
- `or`
- `not`

Observaciones: Cabe revisar los resultados de operaciones lógicas:

- True and True = True
- True and False = False
- True or True = True
- True or False = True
- not False = True
- not True = False

Observe los siguientes ejemplos:

[10]:
```
print((5 < 10) and 4 != 4) # True and False = False
print((3 * 5) < 2 ** 4 or 4 < 5) # False or True = True
print(not (20 > 20.1)) # not False = True
```

```
False
True
True
```

Operadores de asignación aumentada

Estos operadores se utilizan para simplificar una operación aritmética. Reasignan un valor, dado un valor base. La sintaxis es la siguiente:

- `+=`
- `-=`
- `*=`
- `/=`
- `//=`
- `%=`
- `**=`

Los siguientes ejemplos usan el valor base $x = 10$, y a partir de él se reasigna su valor dependiendo de la expresión matemática.

[42]:
```
x = 10
x += 1 # Suma 1
print(x)
x -= 2 # Resta 2
print(x)
x *= 10 # Multiplica por 10
print(x)
x /= 10 # Divide entre 10
print(x)
x **= 2 # Eleva al cuadrado
print(x)
```

```
11
9
90
9.0
81.0
```

2.4. Operaciones y métodos con cadenas de texto

Al igual que con números enteros y flotantes, con cadenas de texto también se realizan operaciones de suma y multiplicación. Veamos los siguientes ejemplos.

- *Concatenar*: Las cadenas de texto se pueden concatenar (unir) con el operador (+) y pueden repetirse con el operador (*).

[9]:
```
print('Para' + 2*'le' + 'pipedo')
```

```
Paralelepipedo
```

Concatenar variable y un literal

[9]:
```
p = 'Mayo'
print(p + 'nesa')
```

```
Mayonesa
```

Además, las cadenas de texto se pueden aplicar distintos métodos como, por ejemplo, convertir de minúscula en mayúscula, o saber la longitud de la cadena.

- `len`: Devuelve la longitud de la cadena de texto

[21]:
```
long = 'sdfghjklgfdwertyuiopxcvbnm'
len(long)
```

[21]: 26

- `upper`: Transforma el texto de minúscula en mayúscula

[33]:
```
saludo = 'buenos días'
saludo.upper()
```

[33]: 'BUENOS DÍAS'

- `capitalize`: Cambia en mayúscula la primera letra

[36]:
```
saludo = 'buenos días'
saludo.capitalize()
```

[36]: 'Buenos días'

Indexación

La indexación en Python es una forma de referir los elementos individuales dentro de un iterable por su posición. Un iterable es un objeto sobre el que se puede iterar; es decir, recorrer por todos los diferentes elementos o caracteres del objeto. Ejemplos de objetos iterables son las listas, las cadenas de texto, las tuplas, etc. Entonces, con la indexación se accede directamente a los elementos que elijas dentro de un iterable. Por otro lado, es importante resaltar que en Python los objetos están 'indexados desde cero', lo que quiere decir que el conteo de posiciones comienza en cero.

Para indexar o acceder a los elementos que se soliciten de un objeto iterable, primero se escribe el nombre de la variable seguido por corchetes (`var[ ]`), en los que llamaremos a una posición o varias posiciones. Tener en cuenta que si se requiere el primer elemento, se debe solicitar la posición 0 (`var[0]`). Ahora, si se requiere acceder desde el primer al quinto elemento, se debe escribir como última posición solicitada el 5 (`var[0:5]`); dado que la indexación toma una posición anterior a la última posición solicitada (0, 1, 2, 3, 4 ->1er elemento, 2do elemento, 3er elemento, 4to elemento, 5to elemento).

Como ya se mencionó, los objetos *string* son iterables, por lo que se pueden indexar. Veamos el siguiente ejemplo.

[14]:
```
w = 'Experimental'
print(w[0]) # se solicita el primer carácter, el cual está en la␣
 ↪posición 0
print(w[5]) # se solicita el sexto carácter, el cual está en la␣
 ↪posición 5
print(w[-1]) # se solicita el último carácter
print(w[0:5]) # se solicita los 5 primeros caracteres, los cuales␣
 ↪encontramos desde la posición 0 a la 4 (5 no incluye)
```

```
E
i
l
Exper
```

Observación: Las cadenas de texto son inmutables. Por eso no se puede modificar la letra de una posición del texto. Veamos que sale error.

[17]:
```
w[0] ='e'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call␣
 ↪last)
Input In [17], in <cell line: 1>()
----> 1 w[0] ='e'

TypeError: 'str' object does not support item assignment
```

Entonces, si se quiere cambiar una letra de la cadena, lo mejor es crear una nueva

cadena de texto.

2.5. Listas

En la sección anterior hemos visto en detalle tipos de datos que solo tienen un elemento. No obstante, como ya dijimos hay estructuras más complejas que agrupan varios tipos de datos. Las listas son un tipo de objeto compuesto, el cual es el más versátil.

Creando listas

Una lista se compone de valores o elementos separados por comas, los cuales están entre corchetes [ ]. Aunque las listas pueden contener valores de distintos tipos, generalmente son del mismo tipo debido a que a menudo este objeto es utilizado para iterar. Veamos los siguientes ejemplos.

Lista vacía

[40]:
```
lst = []
```

Agregando valores

[40]:
```
lst = [1, 2, 3, 5, 8]
print(lst)
```

```
[1, 2, 3, 5, 8]
```

Elementos de diferentes tipos

[40]:
```
dt = ['exp', 2, 4, True]
print(dt)
```

```
['exp', 2, 4, True]
```

Operaciones y métodos con las listas

Del mismo modo que con los anteriores objetos, con las listas también podemos realizar operaciones como de suma (concatenar) o multiplicación. Veamos los siguiente ejemplos.

- *Concatenar*: Con el operador de suma (+) agregamos o unimos horizontalmente las listas.

[46]:
```
lst = [1, 2, 3, 5, 8]
lst_1 = [2, 3, 5, 6, 8, 10]
lst_total = lst + lst_1
lst_total
```

[46]: `[1, 2, 3, 5, 8, 2, 3, 5, 6, 8, 10]`

- *Multiplicación:* el operador de multiplicación (*) repite una lista n veces.

[61]:
```
n = 10
base = [1, 2] * n
print(base)
```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Indexación

Como ya se mencionó, las listas también son objetos iterables, por lo que se pueden indexar. Veamos un ejemplo.

[43]:
```
lst = [1, 2, 3, 5, 8]
print(lst[0]) # se solicita el primer elemento, el cual está en la␣
 ↪posición 0
print(lst[0:3]) # se solicitan los 3 primeros elementos, los cuales␣
 ↪están de la posición 0 a la posición 2 (3 no incluye)
print(lst[1:]) # se solicita desde el segundo al último elemento, los␣
 ↪cuales están desde la posición 1 hasta la última
print(dt[-1]) # se solicita el último elemento
```

```
1
[1, 2, 3]
[2, 3, 5, 8]
8
```

Asimismo, la indexación permite realizar operaciones con los elementos de las listas. Veamos el siguiente ejemplo.

[62]:
```
# Operaciones con los elementos numéricos de una lista
print(lst_total)
print(lst_total[1] * lst_total[2] * lst_total[3]) # 2*3*5
```

```
[1, 2, 3, 5, 8, 2, 3, 5, 6, 8, 10]
30
```

Modificando listas

En contraste con los *strings*, las listas pueden ser modificadas después de su creación debido a que son objetos mutables. Veamos los siguientes ejemplos.

Cambiando un elemento utilizando la indexación.

[75]:
```
frutas = ['plátano', ' manzana', ' uva', 'mando']
frutas[-1] = 'mango' # Corregí el error de tipeo de la palabra
print(frutas)
```

```
['plátano', 'manzana', 'uva', 'mango']
```

Borrando los elementos de una lista según la posición.

[5]:
```
lista = [1, 2, 3, 5, 8, 2, 3, 5, 6, 8, 10]
del lista[3:8] # se borrarán [5, 8, 2, 3, 5, 6]
print(lista)
```

Copiando una lista.

[ ]:
```
lst_a = [1, 2, 3, 4]
lst_d = lst_a.copy()
print(lst_d)
```

```
[1, 2, 3, 4]
```

Métodos que modifican a las listas

- `append`: Añade un elemento al final de la lista.

[76]:
```
frutas.append('naranja')
print(frutas)
```

```
['plátano', ' manzana', ' uva', 'mango', 'naranja']
```

- `extend`: Solo acepta un único argumento iterable (como una lista). Es equivalente a la concatenación de listas con el signo '+'.

[77]:
```
mas_frutas = ['fresa', 'mandarina']
frutas.extend(mas_frutas)
print(frutas)
```

```
['plátano', ' manzana', ' uva', 'mango', 'naranja', 'fresa',␣
↪'mandarina']
```

- `insert`: También inserta argumentos, pero se puede elegir la posición en la que se agregará.

[7]:
```
frutas = ['plátano', ' manzana', ' uva', 'mando']
frutas.insert(2,'fresa')
print(frutas)
```

```
['plátano', ' manzana', 'fresa', ' uva', 'mando']
```

- `pop`: Eliminamos un argumento.

[8]:
```
frutas.pop(2)
print(frutas)
```

```
['plátano', ' manzana', ' uva', 'mando']
```

- `reverse`: Ordena los argumentos de la posición mayor hasta la posición menor.

[86]:
```
lst_a = [2, 1, 4, 6, 5, 8, 9, 10]
lst.reverse()
lst
```

[86]:
```
[10, 9, 8, 5, 6, 4, 1, 2]
```

- `sort`: Ordena de menor a mayor los argumentos numéricos.

[88]:
```
lst_a = [2, 1, 4, 6, 5, 8, 9, 10]
lst.sort()
lst
```

[88]:
```
[1, 2, 4, 5, 6, 8, 9, 10]
```

- `list()`: Declara una lista.

[8]:
```
juegos = list()
# Insertamos algunos juegos a la lista creada
juegos.append('Cournot')
juegos.append('Bienes_publicos')
juegos.append('Bertrand')
juegos.append('Dictador')
juegos.append('Dilema_del_prisionero')
print(juegos)
```

```
['Cournot', 'Bienes_publicos', 'Bertrand', 'Dictador',␣
 ↪'Dilema_del_prisionero']
```

Métodos que modifican a las listas numéricas

También existen otros métodos para listas numéricas que se definen no con la sintaxis usual: *var.nombremetodo*, sino con *nombremetodo(var)*: sum(lista), min(lista), max(lista) y len(lista)

[10]:
```
lst = [2, 1, 4, 6, 5, 8, 9, 10]
print(sum(lst)) # retorna la suma de todos los valores de una lista
print(min(lst)) # retorna el mínimo valor de una lista
print(max(lst)) # retorna el máximo valor de una lista
print(len(lst)) # retorna número total de argumentos de una lista
```

```
45
1
10
8
```

El tipo de objeto `range`

A pesar de que `range` es un tipo de objeto propio, es conveniente explicarlo en este subcapítulo porque este objeto se define como una lista, la cual, a diferencia de las listas que hemos visto anteriormente, es inmutable y contiene números enteros en sucesión aritmética. Recordar que inmutable significa que, a diferencia del objeto lista, el objeto range no se puede modificar. Asimismo, una sucesión aritmética es aquella en la que la diferencia entre dos números consecutivos es siempre la misma.

Para crear un `range` se utiliza la palabra *range* y entre paréntesis se introduce uno, dos o tres argumentos numéricos. Si se usa solo un argumento, se escribe `range(n)`, el cual crea una lista inmutable de n números enteros consecutivos que empieza en *0* y termina en *n-1*. Si se usan dos argumentos, se escribe `range(m, n)`, el cual crea una lista inmutable de números enteros consecutivos que empieza en *m* y termina en *n-1*. Por último, si se usan tres argumentos, se escribe `range(m, n, p)`, el cual crea una lista inmutable de números enteros que empieza en *m* y acaba justo antes de igualar o superar a *n*. El argumento *p* indica que los valores van a ir incrementando de *p* en *p*.

En resumen, en `range(m, n, p)` los tres argumentos son:

- m: indica el primer número
- n: indica el último número, el cual no alcanza nunca
- p: indica los pasos de la progresión aritmética

Por otro lado, antes de ver algunos ejemplos, es importante mencionar que para ver los valores del `range` es necesario convertirlo a lista usando la función `list()`.

Con un argumento:

[12]:
```
w = range(11)
print(list(w))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Con dos argumentos:

[12]:
```
y = range(8, 15)
print(list(y))
```

```
[8, 9, 10, 11, 12, 13, 14]
```

Con tres argumentos:

[12]:
```
z = range(5, 20, 2)
print(list(z))
```

```
[5, 7, 9, 11, 13, 15, 17, 19]
```

2.6. Diccionarios

Otro tipo de objeto útil es el *diccionario*. A diferencia de otras secuencias que se indexan con un rango numérico, los diccionarios se indexan con claves o *keys* que pueden ser de cualquier tipo inmutable. Lo último quiere decir que podemos elegir cadenas o números como claves, dado que son inmutables.

Creando diccionarios

Un diccionario almacena valores en pares y se define como una secuencia de *clave: valor*, separadas por comas y encerradas entre llaves { }. Alternativamente, es posible

crear un diccionario usando *dict()*.

El siguiente ejemplo muestra cómo crear un diccionario utilizando llaves.

[17]:
```
b = {'Maria': 23, 'Juan': 24, 'Miguel': 34}
print(b)
```

```
{'Maria': 23, 'Juan': 24, 'Miguel': 34}
```

Como se señalo, la segunda forma de crear diccionarios es con el comando *dict()*. Este formato de diccionario se encuentra en las plantillas de código de oTree. Veamos el siguiente ejemplo que contiene la misma secuencia de valores que el ejemplo anterior.

[17]:
```
a = dict(Maria = 23, Juan = 24, Miguel = 34)
print(a)
```

```
{'Maria': 23, 'Juan': 24, 'Miguel': 34}
```

También es posible que un valor sea una lista de nombres. En el siguiente ejemplo la clave *secuencia* contiene como valor (o representa) una lista de nombres (*Intrucciones, 'el juego', Información de pago*).

Usando { }

[20]:
```
juego1 = {'nombre': 'guess_two_thirds', 'nombre_desplegado': 'Guess 2/3␣
 ↪of the Average', 'secuencia': ['Instrucciones',␣
 ↪'guess_two_thirds','Información_pago']}
print(juego1)
```

```
{'nombre': 'guess_two_thirds', 'nombre_desplegado': 'Guess 2/3 of the␣
 ↪Average',
'secuencia': ['Instrucciones', 'guess_two_thirds', 'Información_pago']}
```

Usando dict

[20]:
```
juego2 = dict(nombre = 'public_goods', nombre_desplegado = 'Public␣
 ↪Goods Game', secuencia = ['Instrucciones',␣
 ↪'public_goods','Información_pago'])
print(juego2)
```

```
{'nombre': 'public_goods', 'nombre_desplegado': 'Public Goods Game',
'secuencia': ['Instrucciones', 'public_goods', 'Información_pago']}
```

Operaciones con los diccionarios

A continuación, veremos algunas operaciones que podemos realizar con los diccionarios. Así como las listas, los diccionarios también son objetos mutables, por lo que pueden ser modificados después de su creación.

Llamando al valor de un key.

[51]:
```
a['Maria']
```

[51]: 23

Agregando un par *clave:valor*.

[52]:
```
a['Julia'] = 25
a
```

[52]: {'Maria': 23, 'Juan': 24, 'Miguel': 34, 'Julia': 25}

Eliminando un elemento del diccionario.

[53]:
```
del a['Maria']
a
```

[53]: {'Juan': 24, 'Miguel': 34, 'Julia': 25}

Listando a las claves.

[55]:
```
list(a)
```

[55]: ['Juan', 'Miguel', 'Julia']

2.7. Sentencias condicionales y sentencias en bucle

En esta sección se desarrollarán las sentencias if, else, elif, for, while, break, continue y pass. Cada uno de ellas es muy útil para la elaboración de funciones. como se verá más adelante.

Sentencias condicionales

La sentencia if y else

Sintaxis:

```
if condición <booleano>:
        <ejecuta sentencia condicionada>
else:
        <otra sentencia condicionada>
```

En palabras, la sintaxis básica quiere decir:

Si la condición evaluada es verdadera, entonces se ejecuta la primera sentencia. Caso contrario, se ejecuta la segunda sentencia. Veamos el siguiente ejemplo, en el que queremos crear una sentencia que determine si una variable ('x') es menor o mayor a 50.

[97]:
```
x = 34

if x < 50:
    print('x es menor a 50')
```

```
else:
    print('x es mayor a 50')
```

```
x es menor a 50
```

La sentencia if agregando elif

Los `elif` van antes del `else` y evalúan cualquier otra condición cuando no se cumple en la condición del `if`. Cabe indicar que puede haber tantos `elif` como condiciones adicionales se requiera.

Sintaxis:

```
if condición <booleano>:
        <ejecuta sentencia condicionada>
elif condición <booleano>:
        <ejecuta sentencia condicionada>
else:
        <otra sentencia condicionada>
```

Veamos el siguiente ejemplo.

[98]:
```
x = 100

if x < 50:
    print('x es menor a 50')
elif x < 100:
    print('x es menor a 100')
elif x == 100:
    print('x es igual a 100')
else:
    print('otro')
```

```
x es igual a 100
```

Las sentencias en búcles

La sentencia for

La sentencia for de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia.

Sintaxis básica:

```
for valor in <secuencia de valores>:
        <ejecuta sentencia>
```

En palabras, para cada elemento de una secuencia de valores, ejecuta la sentencia indicada. Veamos un ejemplo utilizando el objeto lista.

[102]:
```
lista = [1, 3, 5, 6, 7]
```

```
for i in lista:
    i += 1
    print(i)
```

```
2
4
6
7
8
```

Por otro lado, otro objeto secuencial que es frecuentemente utilizado con la sentencia for es `range`. Es otra manera de poder observar los valores que contiene un objeto `range`. Veamos el siguiente ejemplo.

[27]:
```
for i in range(1, 20, 2):
    print(i)
```

```
1
3
5
7
9
11
13
15
17
19
```

La sentencia while

La sentencia `while` indica que, mientras la condición sea verdadera, se ejecuta la sentencia.

Sintaxis:

```
while condición:
        <ejecuta sentencia>
```

Veamos el siguiente ejemplo que usa la sentencia `while` para sumar los primeros 9 números positivos.

[11]:
```
suma = 0
x = 0

while x < 10:
    suma += x
    x += 1
print(x) # cuando x llega a 10 se detiene la acción de sumar
print(suma) # 1+2+3+...+9
```

```
10
45
```

La sentencia break

La sentencia `break` se utiliza cuando se requiere que una condición termine anticipadamente a lo establecido en un bucle. Veamos el siguiente ejemplo utilizando `while`, en el que la condición debe detenerse cuando la suma supere el número 30.

[17]:
```
suma = 0
x = 0

while x < 10:
    suma += x
    x += 1
    if suma > 30:
        break

print(x) # cuando x es 9, la suma es mayor a 30 (es 45), por tanto ya␣
 ↪no se utiliza
print(suma)
```

```
9
36
```

La sentencia continue

La sentencia `continue` se utiliza junto con una condición. Indica que el programa ignore la instrucción posterior si se cumple dicha condición, y continúe con la ejecución del programa. En el siguiente ejemplo se puede apreciar el uso de `continue` utilizando el comando `for`. Este bucle ignora las vocales en la lista letters.

[22]:
```
letters = list('abdegfuopit')
print(letters)
vocales = ['a','e','i','o','u']

for n in letters:
    if n in vocales:
        continue
    print(n, 'no es una vocal')
```

```
['a', 'b', 'd', 'e', 'g', 'f', 'u', 'o', 'p', 'i', 't']
b no es una vocal
d no es una vocal
g no es una vocal
f no es una vocal
p no es una vocal
t no es una vocal
```

La sentencia pass

Normalmente, se usa para crear clases (ver sección 2.9) en su mínima expresión. Se puede usar cuando una sentencia es requerida solo por su sintaxis. Por ejemplo:

```
class MyEmptyClass:
    pass
```

2.8. Funciones

Funciones predeterminadas frecuentes

Las funciones predefinidas de Python o *Built-in functions* son de gran utilidad porque permiten realizar tareas frecuentes sin necesidad de que uno mismo cree la función. Por ejemplo, ya hemos visto algunas funciones predeterminadas como `print()` o `len()`, pero existen también otras como `round()` o `input()`, las cuales a continuación se explicaran.

***La función** `round()`*

Retorna al número redondeado a la cantidad de decimales solicitada después de la coma. Utiliza dos argumentos: el primero te pide el número a la que se aplicará la función; el segundo escribe un número indicando la cantidad de decimales con la que se quedará después de la coma. Veamos un ejemplo.

[4]:
```
x = 3.141596852
# Redondear a 3 decimales
print(round(x, 3))
# Redondear a un número entero
print(round(x, ))
```

```
3.142
3
```

Veamos otro ejemplo, en el que se obtiene el precio total con impuestos redondeado a 3 decimales.

[32]:
```
tasa = 16.5/100
precio = 99.99
impuesto =  precio*tasa
total_precio = impuesto + precio
round(total_precio, 3)
```

[32]:
```
116.488
```

***La función** `input()`*

Esta función es utilizada en general cuando se necesita que el usuario introduzca alguna información, como su edad o nombre. El programa se detendrá hasta que el usuario haya introducido la información y finalizado con la tecla de retorno 'Enter'.

En este primer ejemplo, se solicita el nombre del usuario.

[2]:
```
a = input('Introduzca su nombre: ')
print(a)
```

```
Introduzca su nombre: María
María
```

En este segundo ejemplo, se solicita la edad del usuario.

[1]:
```
b = input('Introduzca su edad: ')
print(b)
```

```
Introduzca su edad: 22
22
```

Funciones definidas propias del usuario

La función consiste en una serie de sentencias (como las que ya hemos visto) referenciadas bajo un nombre, la cual puede ser utilizada cada vez que se requieran ejecutar la tarea realizada por la función.

El esquema detallado de una función es el siguiente:

```
def nombre(param1, param2):

        '''
        El docstring de la función
        Describe qué hace la función
        Inputs:
            param1: tipo de variable
            param2: tipo de variable
        Output:
            resultado: tipo de variable
        '''

        instr1
        instr2
        instr3
        ...

        return resultado
```

- Para definir una función se utiliza la palabra clave *def*. Luego, se coloca el nombre de la función.
- Luego del nombre, entre paréntesis se especifican los parámetros de la función, que son los inputs.
- Este encabezado termina con dos puntos.
- A continuación, se puede agregar como primera instrucción una cadena de texto que describe la documentación de la función.
- Lo siguiente es el **cuerpo** de la función o las instrucciones de cómo serán utilizados los parámetros de la función.

- Al final la sentencia *return* antecede a lo que arrojará la función.

En el siguiente ejemplo se define una función que crea una tabla de multiplicación de un número dado con los números desde 0 hasta el 9.

[37]:
```
def tabla_del(num):
    '''
    Tabla de multiplicar de un número hasta el 9
    Input:
        num: int
    Output:
        resultado: lista de números
    '''
    resultado = []
    for i in range(10):
        resultado.append(num*i)

    return resultado
```

Definida ya la función, se accede a ella utilizando el nombre de la función `tabla_del()` y agregamos el *input* que queramos.

[37]:
```
# Llamando a la función con input 4
tab_cuatro = tabla_del(4)
tab_cuatro
```

[37]:
```
[0, 4, 8, 12, 16, 20, 24, 28, 32, 36]
```

- Diferencia entre parámetros y argumentos:

Los **parámetros** son los nombres de las variables a la hora de definir la función. En el caso de *tabla_del* es `num`.
Los **argumentos** son los valores que toman los parámetros cuando llamamos a una función. Por ejemplo, cuando `num = 4`.

Es decir, la función se define con parámetros y se ejecuta con argumentos.

- Alcance de una variable

  - Alcance global : los parámetros de una función tienen un alcance global. Es decir, que fue asignada *fuera* de una función y su tiempo de vida se dará mientras corra el programa.

  - Alcance local: es una variable que está definida dentro de la función y no podemos acceder a ella desde fuera de la misma. Solo existe cuando la función es llamada.

    Veamos el siguiente ejemplo, en el que definimos una función que halle la potencia y suma utilizando dos parámetros:

[64]:
```
def potencia_y_suma(a, b):
    '''
    Eleva número "a" a la potencia "b".
    Insumo (input):
        a: número
        b: número
    Producto (output):
        resultado: un número + c
    '''
    d = 10
    resultado = a ** b + d
    return resultado
```

A continuación, se crean 2 variables que serán utilizadas como argumentos de la función anteriormente definida.

[65]:
```
a = 2
b = 5
potencia_y_suma(a, b)
```

[65]: 42

A continuación, se llama a la variable *a*:

[66]:
```
a # Tiene alcance global
```

[66]: 2

Se observa que la variable *a* (y *b*) tiene un alcance global porque fue asignada fuera de la función y su tiempo de vida permanece fuera de la función. En cambio, si queremos acceder a la variable *d* no podemos porque esta tiene un alcance local. Es decir, está definida dentro de una función y solo existe cuando esta es llamada. Entonces, al llamar a la variable *d* fuera de la función arroja *NameError*.

[67]:
```
d # Tiene alcance local
```

```
---------------------------------------------------------------------
NameError                                 Traceback (most␣
 ↪recent call last)
Input In [67], in <cell line: 1>()
----> 1 d

NameError: name 'd' is not defined
```

2.9. Clases y métodos

Python es un lenguaje de programación orientado a objetos; es decir, se basa en el concepto de clases y objetos. Los objetos son una abstracción de los datos que tienen una representación interna, a la cual llamaremos **atributos de datos**, y una interfaz para interactuar con el objeto a través de **métodos o atributos de procedimiento**. Todo en Python es un objeto y es de un tipo específico. En esta última sección, se presenta una breve introducción al tema de clases en Python.

¿Qué son las clases?

Las clases son utilizadas para crear nuevos tipos de objetos. Definamos a las clases como una plantilla que empaqueta atributos para, a partir del nuevo tipo de objeto, crear instancias. Una instancia significa crear un objeto de la clase definida, lo cual se logra creando variables del objeto. Por ejemplo, `n = [1, 2, 3, 4]` es una instancia de la clase lista.

Al crear una clase se debe definir el nombre de la clase y sus atributos (características y métodos). Por ejemplo, alguien implementó un código para crear la clase lista. Así se creo el tipo de objeto lista que conocemos. Se puede crear tantos objetos como quieras y realizar todas las funciones u operaciones que hayas definido en la clase. Por ejemplo: `list.append` es un método de la clase lista como se explicó en la sección 2.5.

La estructura básica de una clase es la siguiente:

```
class Class_name(object):

    # Definir atributos aquí

    <statement_1>
    .
    .
    .
    <statement_n>
```

- Para definir una clase se utiliza la palabra clave *class*.
- Se coloca un nombre para el nuevo tipo de objeto creado: *Class_name*.
- Luego del nombre, entre paréntesis puede ir el comando *object* para indicar que se está creando un objeto. No obstante, ese espacio puede dejarse vacío.
- Este encabezado termina con dos puntos.
- El contenido posterior es variable. Los atributos son datos y procedimientos que "pertenecen" a la clase:

    1. Atributos de datos: pensar en los datos como otros objetos (números, cadenas de texto, listas, booleanos) que van a componer el nuevo tipo de objeto. Estos datos caracterizan al objeto.

2. Atributos de procedimiento o métodos: son funciones que definen la forma cómo van a interactuar los objetos del tipo creado. Ciertamente, primero se debe crear el objeto, y luego el método.

Creando un tipo de objeto

En esta subsección se define un tipo de objeto llamado *Contribuyente*. Esta clase se va a basar en un contribuyente, el cual está caracterizado por sus nombres y apellidos, DNI o RUC e ingresos anuales.

- **Creando el tipo de objeto *Contribuyente* en su mínima expresión**

[1]:
```
class Contribuyente(object):
    pass
```

Una vez definida la clase, se puede instanciar en su mínima expresión. Para ello solo es necesario crear variables del tipo contribuyente.

[2]:
```
x = Contribuyente() #Instancia
y = Contribuyente() #Instancia
```

- **Definiendo atributos de datos del tipo de objeto *Contribuyente***

Para definir atributos de datos de una instancia se utiliza la función **__init__**. Esta función especial define atributos de instancias en un **estado inicial**. Es decir, los atributos de instancias tienen que especificarse al momento de instanciar. Los atributos de instancia son características únicas para cada objeto creado.

Para nuestro tipo de objeto *Contribuyente* vamos a definir como atributos de datos el nombre, el primer apellido, el RUC o DNI y los ingresos anuales.

[3]:
```
class Contribuyente(object):
    def __init__(self, nombre, apellido, dni_RUC,␣
 ↪ingresos_anuales):
        self.nombre = nombre
        self.apellido = apellido
        self.dni_RUC = dni_RUC
        self.ingresos_anuales = ingresos_anuales
```

Explicación de la función __init__:

- Como toda función, __init__ comienza con el comando `def`.
- Luego, entre paréntesis se escribe el comando `self`. Ese es un parámetro que siempre se utiliza para que luego de instanciar se pueda acceder a los datos que se definan para cada instancia. Por ejemplo, más adelante verá que la instancia 'M' reemplaza a la palabra `self`.
- Además del parámetro anterior, se pueden agregar otros parámetros (atributos de datos) que tomarán valores específicos para cada instancia.

A continuación, se agregarán los atributos de datos a las dos instancias de tipo *Contribuyente* etiquetadas como M y J.

[2]:
```
M = Contribuyente('María', 'Gonzales', 74238266, 30000)
J = Contribuyente('José', 'Torres', 45227234, 50000)
```

Para acceder a los atributos de datos que se han definido de las instancias 'M' y 'J' escribimos el nombre del objeto seguido de un punto y, luego, el atributo que se quiera llamar. Veamos los siguientes ejemplos.

[4]:
```
print(M.nombre)
print(J.dni_RUC)
```

```
María
45227234
```

- **Definiendo un método de la clase *Contribuyente*: suma de ingresos de dos contribuyentes**

  Para sumar los ingresos de dos contribuyentes se va a definir la función llamada *suma_ingresos* que tendrá como parámetros a la palabra *self*, que hace referencia a la instancia misma que se usa como base, y la palabra *other*, que hace referencia a otra instancia diferente a la *self*. Asimismo, como ya se mencionó, la notación del punto se usa para acceder a la data. Veamos el código.

[40]:
```
class Contribuyente(object):
    def __init__(self, nombre, apellido, dni_RUC,␣
 ↪ingresos_anuales):
        self.nombre = nombre
        self.apellido = apellido
        self.dni_RUC = dni_RUC
        self.ingresos_anuales = ingresos_anuales
    def suma_ingresos(self, other):
        ingresos_totales = self.ingresos_anuales + other.
 ↪ingresos_anuales
        return(ingresos_totales)
```

A continuación ilustramos cómo usar el método creado. Primero se crea las instancias, y luego se llama al método. Creando las instancias

[41]:
```
M = Contribuyente('María', 'Gonzales', 74238266, 100000)
J = Contribuyente('José', 'Torres', 45227234, 50000)
```

Existen dos formas de usar el método. La forma convencional:

[43]:
```
# Forma convencional
print(M.suma_ingresos(J))
```

```
150000
```

La otra forma de usar el método:

[43]:
```
# Segunda forma
print(Contribuyente.suma_ingresos(M, J))
```

```
150000
```

- **Otros métodos**

  - Cálculo del impuesto a la renta de cuarta categoría

    Para el cálculo del impuesto de un contribuyente se va definir una función llamada *impuesto*. En Perú, el cálculo del impuesto depende del nivel de ingresos anuales (del cual se aplican descuentos), del valor de la Unidad Impositiva Tributaria (UIT) y de un sistema de tasas según cada tramo del ingreso[3]. A continuación se define el método.

[40]:
```
class Contribuyente(object):
    def __init__(self, nombre, apellido, dni_RUC,␣
 ↪ingresos_anuales):
        self.nombre = nombre
        self.apellido = apellido
        self.dni_RUC = dni_RUC
        self.ingresos_anuales = ingresos_anuales

        def suma_ingresos(self, other):
        ingresos_totales = self.ingresos_anuales + other.
 ↪ingresos_anuales
        return(ingresos_totales)

        def impuesto(self):
        UIT = 4400
        desc = self.ingresos_anuales*0.8
        monto_base = desc - 7*UIT

        if  monto_base < 0:
            print(0)
        elif monto_base > 0 and monto_base <= 5*UIT:
            print(0.08*monto_base)
        elif monto_base > 5*UIT and monto_base <= 20*UIT:
            print(0.08*5*UIT + 0.14*(monto_base-5*UIT))
        elif monto_base > 20*UIT and monto_base <= 35*UIT:
            print(0.08*5*UIT + 0.14*(20*UIT - 5*UIT) + 0.
 ↪17*(monto_base - 20*UIT))
        elif monto_base > 35*UIT and monto_base <= 45*UIT:
```

[3]Para mayor información ver el siguiente enlace: https://www.gob.pe/7318-superintendencia-nacional-de-aduanas-y-de-administracion-tributaria-calcular-el-impuesto-de-cuarta-categoria.

```
            print(0.08*5*UIT + 0.14*(20*UIT - 5*UIT) + 0.
↪17*(35*UIT - 20*UIT) + 0.20*(monto_base - 35*UIT))
        elif monto_base > 45*UIT:
            print(0.08*5*UIT + 0.14*(20*UIT - 5*UIT) + 0.
↪17*(35*UIT - 20*UIT) + 0.20*(45*UIT-35*UIT) + 0.
↪30*(monto_base - 45*UIT))
```

Como ya se conoce, para usar el método, primero se crean las instancias.

[22]:
```
M = Contribuyente('María', 'Gonzales', 74238266, 100000)
J = Contribuyente('José', 'Torres', 45227234, 50000)
```

Existen dos formas de llamar al método.

- Primera forma convencional

[23]:
```
M.impuesto()
```

```
5568.0
```

- Segunda forma

[25]:
```
Contribuyente.impuesto(J)
```

```
736.0
```

- Print

  Por defecto, al usar el comando `print()` de un objeto de la clase *Contribuyente* se muestra lo siguiente:

[29]:
```
M = Contribuyente('María', 'Gonzales', 74238266, 100000)
print(M)
```

```
<__main__.Contribuyente object at 0x0000021578886610>
```

Para que dicho comando nos muestre los valores o atributos de datos de un objeto del tipo *Contribuyente* se debe usar un método especial llamado **__str__**. Este método se usa para escribir en formato de texto lo que se quiere que aparezca al llamar un objeto con el comando `print()`. En el código a continuación se añade este método.

[44]:
```
class Contribuyente(object):
    def __init__(self, nombre, apellido, dni_RUC,␣
↪ingresos_anuales):
        self.nombre = nombre
        self.apellido = apellido
        self.dni_RUC = dni_RUC
        self.ingresos_anuales = ingresos_anuales

    def impuesto(self):
```

```
        UIT = 4400
        desc = self.ingresos_anuales*0.8
        monto_base = desc - 7*UIT

        if  monto_base < 0:
            print(0)
        elif monto_base > 0 and monto_base <= 5*UIT:
            print(0.08*monto_base)
        elif monto_base > 5*UIT and monto_base <= 20*UIT:
            print(0.08*5*UIT + 0.14*(monto_base-5*UIT))
        elif monto_base > 20*UIT and monto_base <= 35*UIT:
            print(0.08*5*UIT + 0.14*(20*UIT - 5*UIT) + 0.
↪17*(monto_base - 20*UIT))
        elif monto_base > 35*UIT and monto_base <= 45*UIT:
            print(0.08*5*UIT + 0.14*(20*UIT - 5*UIT) + 0.
↪17*(35*UIT - 20*UIT) + 0.20*(monto_base - 35*UIT))
        elif monto_base > 45*UIT:
            print(0.08*5*UIT + 0.14*(20*UIT - 5*UIT) + 0.
↪17*(35*UIT - 20*UIT) + 0.20*(45*UIT-35*UIT) + 0.
↪30*(monto_base - 45*UIT))

    def suma_ingresos(self, other1):
        ingresos_totales = self.ingresos_anuales + other1.
↪ingresos_anuales
        return(ingresos_totales)

    # Se añade el método __str__
    def __str__(self):
        return("(" + str(self.nombre) + ',' + ' ' + str(self.
↪apellido) + ',' + ' ' + str(self.dni_RUC) + ',' + ' ' +␣
↪str(self.ingresos_anuales) + ")")
```

Se crea la instancia, y ejecutando el comando `print()` ahora se tiene.

[45]:
```
M = Contribuyente('María', 'Gonzales', 74238266, 100000)
print(M)
```

```
(María, Gonzales, 74238266, 100000)
```

2.10. Ejercicios propuestos

1. Sea la siguiente lista:

   `lista = [1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 40, 50, 60]`

   A partir de ella, realizar las siguientes modificaciones:

   - Añade un elemento al final de la lista

- Borra desde el segundo elemento hasta el quinto elemento
- Agrega un valor numérico en la posición 5
- Crea una nueva lista con los elementos desde la primera posición hasta la quinta.
- Crea variables que muestren la suma, el mínimo, el máximo de los valores de esa lista y la longitud de la lista.

2. Crea un diccionario con los nombres de los integrantes de tu familia (incluyendo tu mascota si es que tuvieras) como los *keys*, y las edades como los valores. A partir de ese diccionario genera las siguientes modificaciones usando los métodos vistos para diccionarios:
   - Elimina dos argumentos
   - Agrega otros dos argumentos con el nombre y edad de dos de tus artistas favoritos
   - Crea una lista llamada names a partir de los *keys* del diccionario
3. Crea un programa que devuelva el ciclo de vida en la que usted se encuentra introduciendo su edad. Para ello tendrá que hacer uso de las sentencias `if, elif y else`. Si la persona que ingresa su edad se encuentra entre 0 a 5 años , que imprima "Infancia", si está entre 6 y 11 años es "Niñez", si esta entre 12 y 18 años es "Adolescencia", si está entre 19 y 35 años es "Juventud", si está entre 36 y 65 años es "Adultez" y si es mayor a 65 años es "Vejez". Si la persona introdujo un número fuera de los rangos establecidos (digamos un número negativo) debe aparecer un mensaje de error advirtiendo que debe introducir números enteros positivos.

   Utilice el siguiente código en el inicio de este ejercicio para que la persona introduzca su edad: `edad=int(input('Ingrese su edad: '))`. Además, se aconseja utilizar el operador lógico and al usar las sentencias condicionales.
4. Utilizando el comando `while` y una variable numérica de valor 10, imprima la suma de los números pares que se encuentran hasta el valor 10.
5. Utilizando el comando `for` e `if y else` imprima solo las letras vocales de un string que contenga todas las letras del abecedario. Use la siguiente variable string:

   ```
   abc = ''abcdefghijklmnñopqrstuvwxyz''
   ```
6. Sentencias en búcles:
   - Crea una lista con los números 10, 20, 30 ... 100 (usar `range`)
   - Utilizando la sentencia `for`, itera sobre los valores de esa lista y que devuelva para cada valor una nueva lista con el cuadrado de cada número, pero que se detenga cuando el cuadrado del número sea mayor a 900. (ayuda: usar `break`)
   - Devolver la lista ordenada con los valores de mayor a menor.

7. Crea una función que arroje una tabla de potencias de 0 al 10 en base a un número que será el parámetro de la función. (Ayuda: seguir el ejemplo visto en 3.7.Funciones). Se recalca que solo se necesita de un parámetro.

8. Defina una función que filtre palabras de una lista según la longitud de las palabras. Llame a esa función *filtrar_palabras()*. Utilice como argumentos una lista de palabras (lista) y un numero entero(n). La función debe retornar una lista con las palabras que tienen más de n caracteres. Si ninguna palabra cumple la condición y la lista no tiene ninguna palabra que imprima "Ninguna palabra cumple la condición".

   Ayuda: Al empezar la función cree una lista vacía, que será la que imprima la función con las palabras filtradas. Utilice un `for loop` junto con un condicional simple (solo `if`) para realizar el filtro. Para el filtro usar los métodos de la lista: `len y append`.

9. Según el ejercicio visto en la sección de Clases, cree un nuevo tipo de objeto que represente una coordenada.

   - La representación interna son dos números separados por una coma: (x, y). Recuerde usar el método especial __str__ para que al momento de usar el comando `print()` se vea la representación de este tipo de objeto.
   - Por otro lado, cree los siguientes métodos: la distancia entre dos coordenadas y la suma entre dos coordenadas.

Capítulo 3

Programación de Experimentos I

3.1. Sobre oTree

Según la documentación[1] y Chen et.al. (2016), oTree es una plataforma de código abierto para programar e implementar experimentos de laboratorio (presencial y en línea) o de campo, o de ambos. El marco de trabajo de oTree está basado en Python y entre las tareas que permite se encuentran las siguientes: encuestas; juegos de estrategias multijugador, tales como dilema del prisionero, bienes públicos y subastas; experimentos conductuales controlados en economía, psicología y áreas relacionadas. De acuerdo con su portal, a la fecha esta plataforma ha sido usada en más 1100 publicaciones académicas.

Existen recursos virtuales que permiten aprovechar al máximo la plataforma. En primer lugar, se encuentra al grupo de Google para oTree, donde se comentan diversas dudas sobre la plataforma[2]. En segundo lugar, oTree ofrece una amplia documentación guía para un mejor manejo de la plataforma[3]. Asimismo, la plataforma brinda una serie de proyectos demostrativos que sirven para guiarse al programar un experimento. Finalmente, como proyecto de código abierto, oTree cuenta con un repositorio en GitHub, en el cual se puede colaborar y descargar experimentos hechos por otros usuarios[4].

Cabe indicar que existe una versión antigua y otra actual de oTree: la 3.x y la 5.x, respectivamente. En este capítulo se detallan los pasos para instalar la versión actual, mientras que en el Apéndice A.3 se hace lo propio con la versión antigua. En este documento se utiliza la versión actual. No obstante, tener la versión antigua es útil, pues muchos experimentos han sido escritos en dicha versión.

El principal objetivo de este capítulo es presentar los elementos básicos de la programación de experimentos en la versión actual de oTree, tomando como ejemplo una encuesta. Luego de presentar de manera general a oTree en esta sección, en la sección 2 se explica paso a paso la instalación de la plataforma oTree. En la sección 3 se crea un

[1] https://otree.readthedocs.io/en/latest/
[2] Link del grupo de Google para unirse: https://groups.google.com/g/otree?pli=1.
[3] https://otree.readthedocs.io/en/latest/
[4] Otree Hub: https://www.otreehub.com/projects/?

proyecto en el que se descarguen las plantillas predeterminadas que existen en oTree. En la sección 4 se menciona la estructura base que presenta el programa oTree. En la sección 5 se describe el localhost de oTree, lo cual permite visualizar el experimento en el servidor local. En las sección 6 se estudia la configuración general de un proyecto. En la sección 7 se explora el código de un aplicativo del proyecto: Survey. En la sección 8 se enseña cómo descargar los datos que se obtiene luego de ejecutar un experimento, así como se describen las principales variables que se encuentran en la descarga. La última sección contiene ejercicios propuestos.

3.2. Pre requisitos e instalación de oTree versión 5

Instalación de oTree

Para esta sección, se asume que se ha instalado Anaconda, y que se tiene familiaridad con el manejo de environments. Si no es el caso, se sugiere revisar el Apéndice A.2. Asimismo, cierta familiaridad con el CMD de Windows es útil, por lo que se recomienda revisar también el Apéndice A.1.

Como se explica en el Apéndice A.2, para crear un environment llamado "Otree5" con la versión 3.9 de Python (que será útil para experimentos), ejecute:

```
conda create -n Otree5 python=3.9
```

Cuando Anaconda le pregunte si desea proceder, presione 'y'. A continuación, active el environment creado con el comando:

```
conda activate Otree5
```

Para instalar la versión actual de oTree, escriba lo siguiente:

```
pip3 install -U "otree>=5.4.0"
```

Es importante no olvidar la última parte ("otree>=5.4.0") ya que con eso asegura descargar la última versión. En este documento se trabaja con esa versión porque es la actual y también porque presenta algunas facilidades al ser una versión compacta y que no depende de Django[5]. Además, una vez que se conoce cómo funciona la versión de oTree 5, entender la versión 3 es más sencillo.

Algunas consideraciones

Para la edición y manejo de códigos se utiliza la plataforma Visual Studio Code[6]. Los conocimientos de Python mínimos incluyen un manejo de listas y diccionarios, así como funciones, clases y estructuras selectivas y repetitivas (temas que se revisan en el capítulo 2).

[5] Es un framework de desarrollo web de código abierto, escrito en Python.
[6] También se puede utilizar otro editor como Pycharm

3.3. Creando su primer proyecto

En esta sección se van a descargar las plantillas de experimentos ya existentes en oTree, tales como el juego de bienes públicos, Cournot o el dilema del prisionero, las cuales se almacenarán en una carpeta o proyecto que se llamará *primer_proyecto*. El objetivo de esta sección es crear nuestro primer proyecto, el cual contendrá aplicaciones predeterminadas de oTree para poder explorar alguno más adelante en este capítulo.

Primero, en su PC cree una carpeta llamada "Experimentos"[7]. Esta carpeta va a contener todos sus experimentos. A continuación, una vez ya activado el enviroment "Otree5" en el Anaconda Prompt, ubíquese en esa carpeta de la siguiente manera:

```
cd C:\Experimentos
```

Luego, crea una carpeta del proyecto en concreto que se llamará "primer_proyecto" de la siguiente manera:

```
otree startproject primer_proyecto
```

Aparecerá la siguiente pregunta:

```
Include sample games? (y or n):
```

A la pregunta *Include sample games?* se responde escribiendo *y*. Esto permitirá descargar diversos ejemplos de juegos o actividades (aplicativos) ya diseñados y otros moldes que facilitarán la programación.

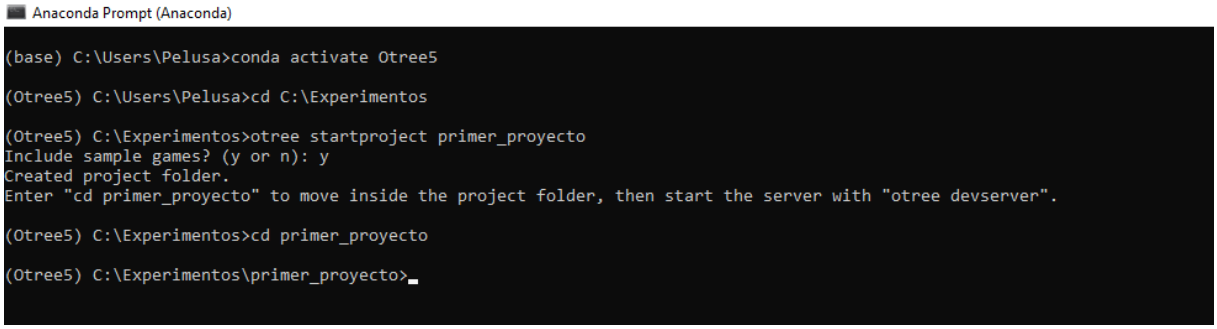
Para comprobar que la carpeta fue creada vaya a la carpeta que hemos llamado "Experimentos" y dentro de esta verifique si se ha creado la carpeta "primer_proyecto". Usted podrá ver que esta contiene una serie de plantillas sobre diversas actividades, entre otros aspectos.

A continuación, procedemos a mover el directorio a nuestra carpeta creada:

```
cd primer_proyecto
```

Si sigue todas las indicaciones hasta ahora, debe quedar así:

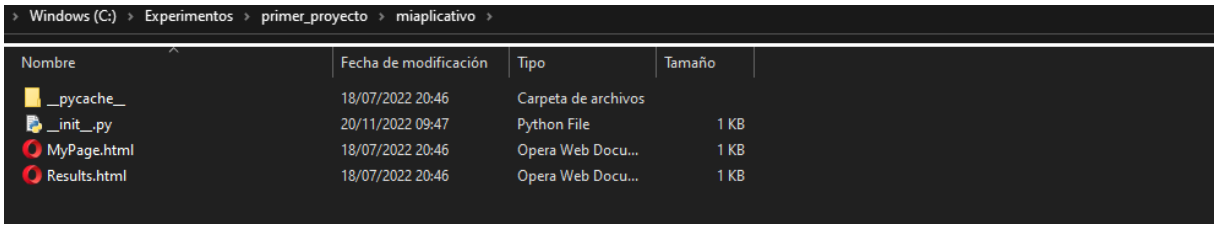
[7] En este caso, la carpeta fue creada en el Disco C.



Por otro lado, para añadir una subcarpeta para algún aplicativo en específico, use el comando *otree startapp*. Cabe señalar que un aplicativo representa una actividad específica, la cual se materializa en una serie de páginas. Ahora, por ejemplo, si quiero llamar a mi subcarpeta "miaplicativo":

```
otree startapp miaplicativo
```

Si se dirige a su proyecto 'primer_proyecto', ahora encontrará una subcarpeta llamada "miaplicativo", como se ve en la siguiente imagen:



Cabe mencionar que dicha subcarpeta contiene algunos archivos base que sirven como plantilla para la creación de un aplicativo nuevo. Esto se detalla en la sección 3.7.

3.4. Estructura de un experimento en oTree

A continuación se describe la estructura jerárquica en la que se programa un experimento. Esta es de especial interés para entender la programación en oTree en general. En particular, sirve para definir las variables que se crean dentro del archivo que configura una actividad, tema que se trata en la sección 3.7.

En oTree se tiene la siguiente estructura principal para cada experimento: sesión, subsesión y páginas. Una sesión es un evento en la que asisten participantes para realizar una serie de actividades (que pueden ser interactivas o no). Estas actividades se agrupan para formar subsesiones; es decir, cada subsesión contiene una o más actividades. De esta manera, una sesión se puede entender como una serie de subsesiones[8]. Asimismo, cada subsesión se materializa en una secuencia de páginas.

[8]Si una actividad tiene más de una ronda (se repite la misma actividad varias veces), cada una de estas también forma una subsesión.

Es oportuno recalcar que, en cuanto a la programación, cada subsesión está formada por una secuencia de aplicativos. Como ya se precisó, en cada aplicativo se programa una actividad específica, y tiene como output un conjunto de páginas. Para cada aplicativo se crea una subcarpeta. Por otro lado, puede ser que una sesión no tenga subsesiones debido a que la programación del experimento no requiere la agrupación de actividades[9]. En este caso, se puede pensar que la única sesión es, a su vez, una subsesión.

Cabe indicar que el número de participantes se determina a nivel de sesión. Si alguna actividad (subsesión) es interactiva, se requiere formar grupos de participantes, a cuyo número oTree denomina número de jugadores por grupo. En una encuesta, como no hay interacción, no hay grupos.

3.5. Descripción del localhost de oTree

A continuación, se habilitará el programa para que lo visualice en el *server* o *host* local de oTree. En general, un servidor remoto permite que diferentes usuarios puedan acceder de manera independiente desde diferentes equipos a un *software* (oTree, en el caso de experimentos). En este caso al ser un servidor local se podrá acceder solo desde el equipo del programador. Como se verá, esta facilidad permite que el programador de experimentos pueda hacer pruebas de su código, antes de llevar a cabo la real ejecución de su experimento desde un servidor remoto, como Heroku, por ejemplo. ''

Para ello, desde el Anaconda Prompt, debe activar el enviroment "Otree5"; y después, se debe ubicar en la carpeta del proyecto de la siguiente manera:

```
cd C:/Experimentos/primer_proyecto
```

Luego, para acceder al servidor de oTree, escriba lo siguiente[10]:

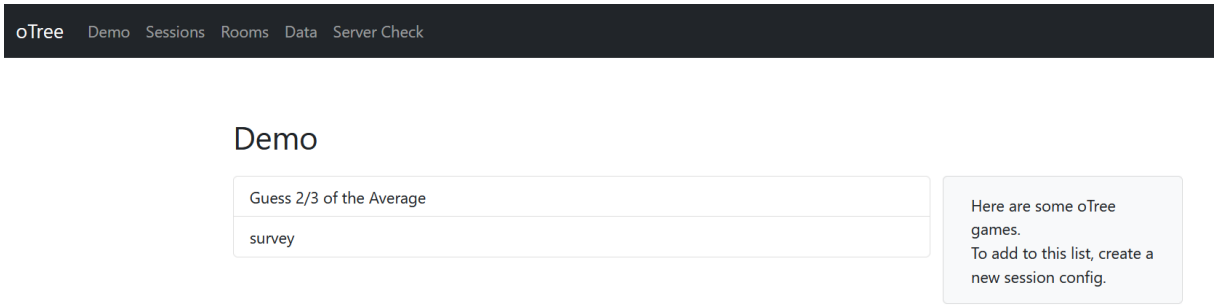
```
otree devserver
```

Le aparecerá un link (texto resaltado), el cual tendrá que copiar y pegar en su navegador.

[9]Como se verá más adelante, este es el caso de una encuesta o el juego de los bienes públicos, entre otros de los juegos predeterminados en oTree.

[10]Es posible que le aparezca el siguiente mensaje: "oTree has been updated. Please delete your database (db.sqlite3)". En caso suceda, dirijase a la carpeta del proyecto y elimine el archivo db.sqlite3. Luego, vuelva a escribir "otree devserver".

Una vez pegado el link en el navegador, aparecerá lo siguiente:



Fuente: Servidor local de oTree

En la barra horizontal de la imagen anterior, se identifican las siguientes entradas:

- Demo: Es la primera ventana que aparece por defecto al abrir el link que oTree proporciona. Como el nombre lo indica, en esta ventana podremos ejecutar actividades de prueba de forma sencilla.
- Sessions: En esta ventana se guardan, archivan o eliminan las sesiones que crearemos en Rooms.
- Rooms: En esta ventana se crean las sesiones y se hace seguimiento a los jugadores, entre otros aspectos relacionados. Sin embargo, también para fines de especificar las características generales (configurar) de una sesión en el navegador. Allí, de acuerdo con lo programado en oTree, podrá elegir la actividad y la cantidad de jugadores correspondientes para que posteriormente se desarrolle el mismo.
- Data: En esta ventana se exportan los datos generados con el experimento.
- Server Check: Aquí se da información relacionada al servidor.

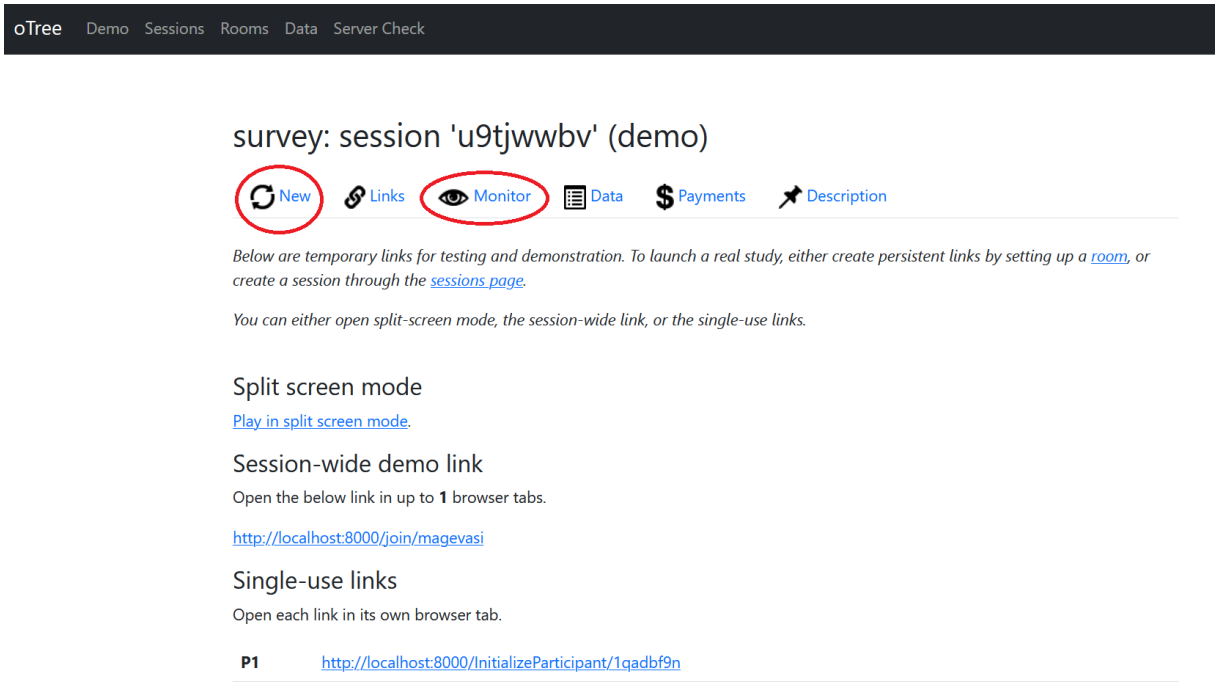
Cabe indicar que las entradas Demo y Data son suficientes para correr el experimento y ver los resultados. Por conveniencia, generalmente, se utiliza solo el Demo y la Data desde el localhost de oTree porque es más rápido para la visualización de cambios en el código del experimento[11].

A continuación, se realizarán breves descripciones y pasos a seguir para acceder a la información útil que proporciona la ventana Demo.

Contenido de la entrada Demo

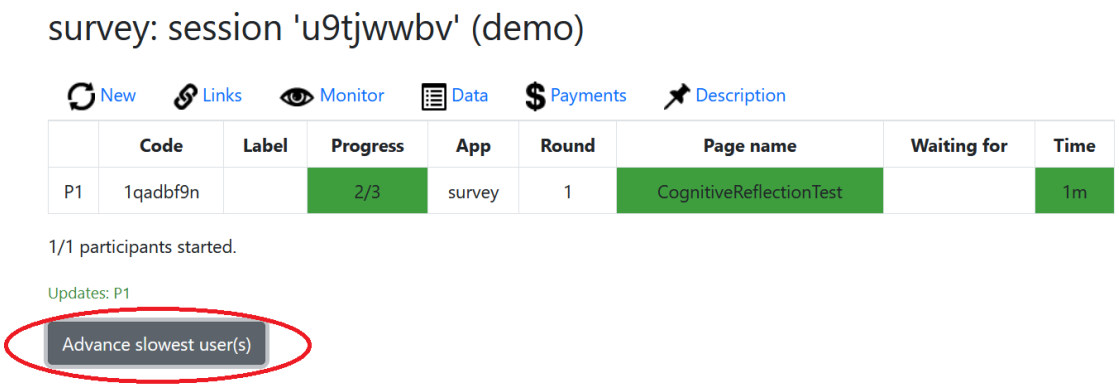
La pestaña "Demo" contiene las sesiones (juegos o encuestas) que han sido configurados (ver la sección 3.6). Por defecto, el proyecto contiene dos: *Guess 2/3 of the average* y *Survey*. Ahora, seleccionando la sesión llamada *Survey* se desplegará la siguiente ventana:

[11]Un bloque importante es el formado por las entradas Sessions, Rooms y Data. Este bloque es de especial utilidad al momento de desplegar y ejecutar el experimento desde un servidor remoto, como la plataforma Heroku



Fuente: Servidor local de oTree

Desde ahí podrá controlar la sesión. Por ejemplo, si selecciona "New", la actividad se reinicia. Ahora, para empezar a ejecutar la sesión, haga click en el último enlace. Como es una encuesta, basta con un participante para que se desarrolle completamente la sesión, por ello solo hay un link[12]. Una vez hecho eso, si selecciona "Monitor" podrá acceder a información actualizada de lo que realiza el encuestado. Esto se visualiza en la siguiente ventana:



Fuente: Servidor local de oTree

En el cuadro verde se observa que el encuestado se encuentra en la página 2 de un total de 3, la cual se llama "CognitiveReflectionTest". Por otro lado, en caso desee probar si la sesión funciona correctamente sin importar los resultados, haga click en "Advance

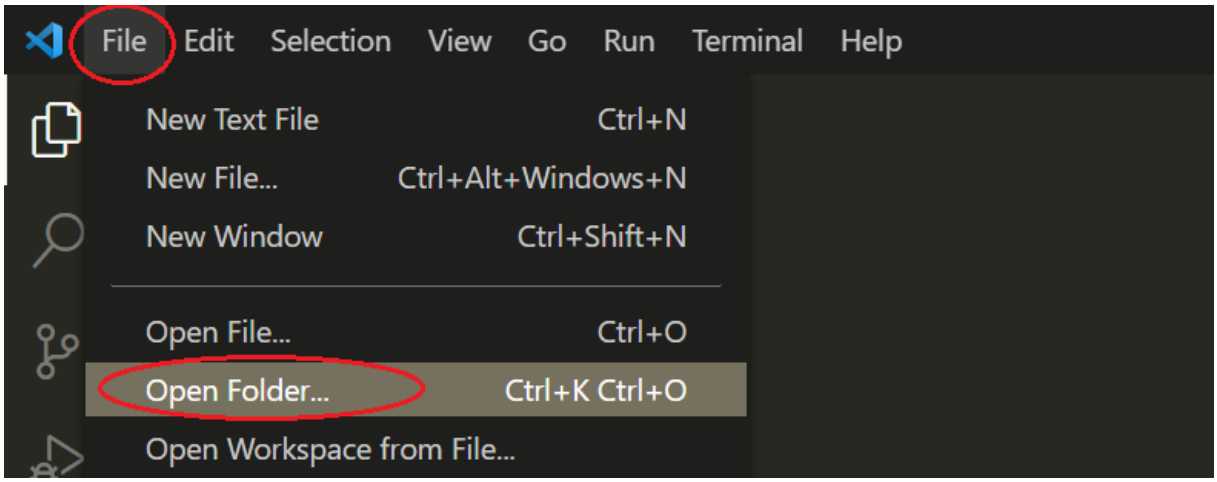
[12]En cambio, si selecciona la sesión *Guess 2/3 of the average* aparecerán 3 links, pues 3 participantes interactúan en este caso.

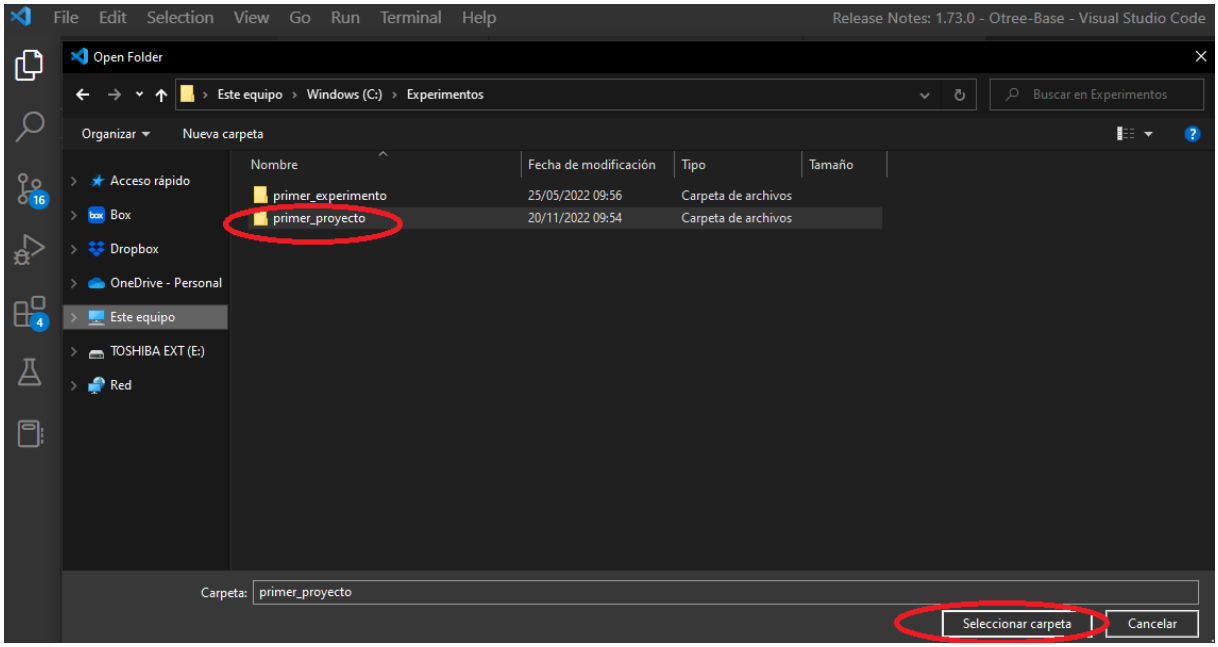
slowest user(s)" cuántas veces sean necesarias hasta terminar el número de páginas. Cabe indicar que de esta manera la actividad avanzará de manera automática.

3.6. Configuración general de un proyecto

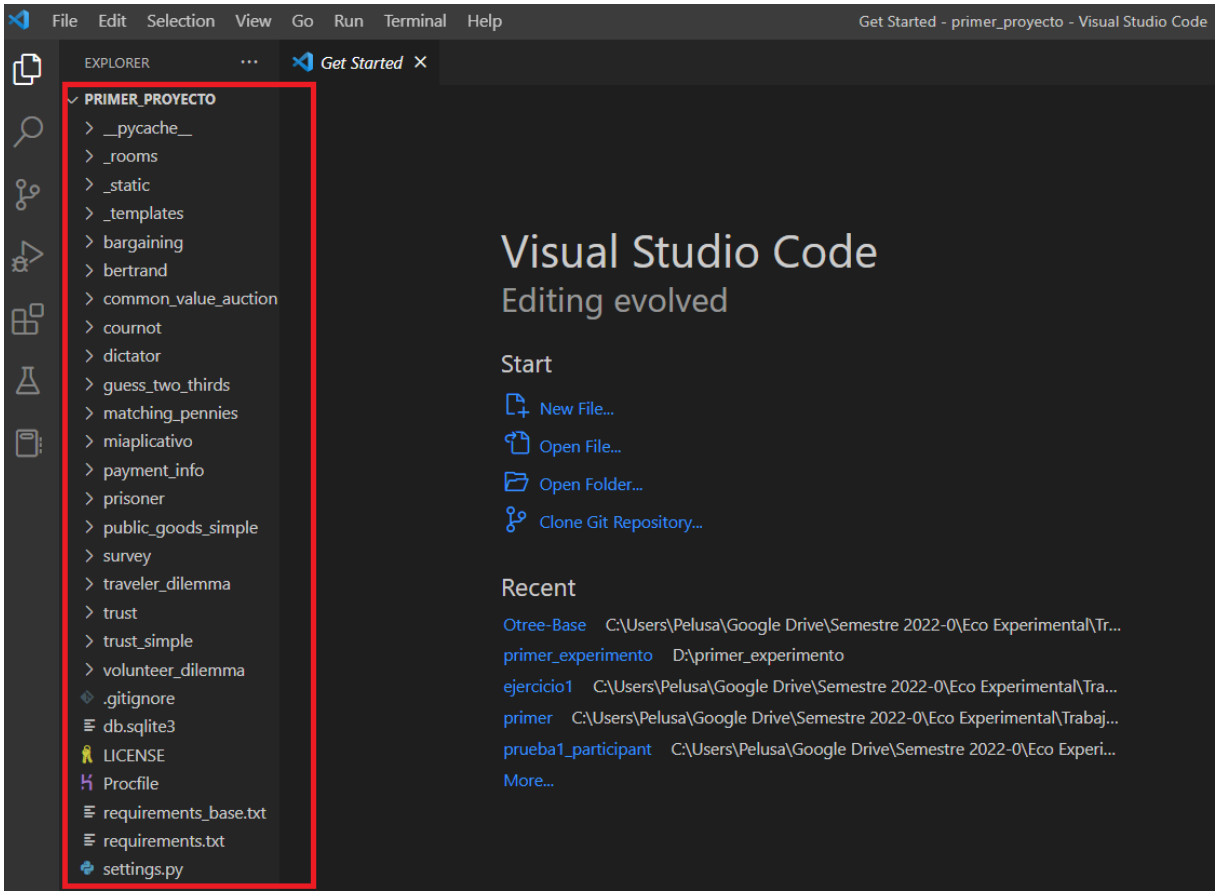
El objetivo principal de esta sección es describir las configuraciones generales de todo el proyecto, las mismas que se programan en el archivo settings.py. Previamente, se describe el contenido (carpetas y archivos) del proyecto en el editor de código.

Dentro del editor de código (VS Code), abra la carpeta "primer_proyecto" de la manera en la que se ve en las siguientes imágenes.





Una vez que abra la carpeta "primer_proyecto", en la parte izquierda, encontrará todos los archivos y carpetas que esta contiene, tal y como se puede observar en la siguiente imagen. Cabe destacar que esta carpeta contiene, en particular, las subcarpetas de todos los aplicativos del proyecto, desde "bargaining" hasta "volunteer_dilemma". Asimismo, en la parte inferior de la imagen se observa que esta carpeta también contiene el archivo *settings.py*, desde el cual se configura todo el proyecto.



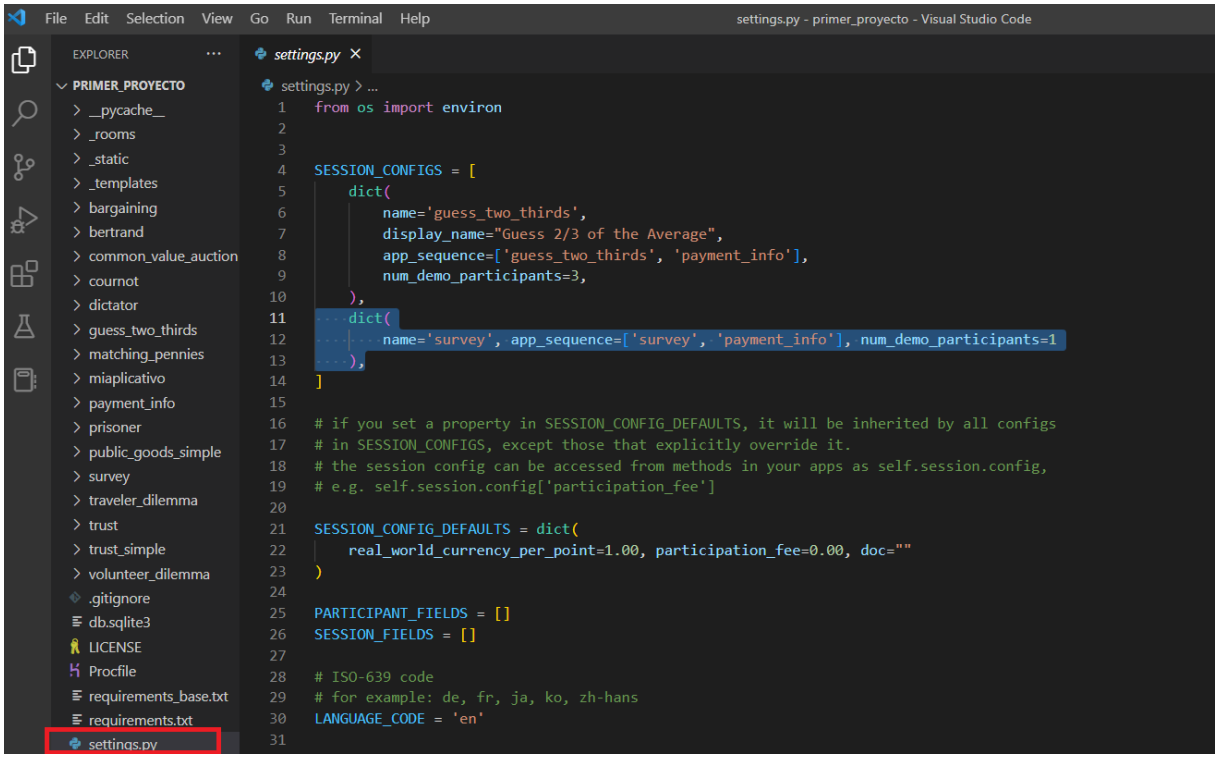
Fuente: Plantillas de oTree

Configuraciones generales del proyecto

Como ya se sabe, las configuraciones generales del proyecto se realizan desde el archivo *settings.py*. En él, se configuran las sesiones, los rooms, el idioma, la moneda real, entre otros. A continuación, se describen las secciones importantes de este archivo.

- SESSION_CONFIGS

  En este apartado se mencionan en diccionarios las sesiones que se quieren agregar al demo en la web.



Fuente: Plantillas de oTree

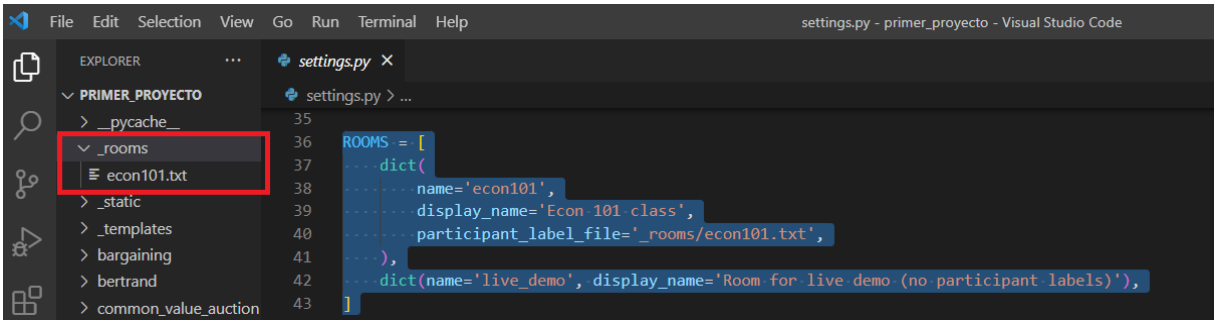
Usualmente, este diccionario solicitará los siguientes *keys* y *values*:

- Name: es el nombre interno que se quiere dar a la sesión.
- Display_name: es el nombre que aparece en el demo. Si no se incluye esta opción en el diccionario, el valor es el mismo al del nombre interno. En el caso de la sesión "survey", como el diccionario sombreado no contiene la opción `display_name`, el nombre público de la sesión será el mismo al nombre interno, es decir, "survey".
- App_sequence: es el orden en el que aparecerán los aplicativos al participante al desarrollarse la sesión. En la sesión llamada "survey", el primer aplicativo será "survey" y el segundo será "payment_info". El primero muestra las páginas de la encuesta propiamente dicha; mientras que el segundo, la información sobre los pagos que corresponden al participante.
- Num_demo_participants: es la cantidad de participantes en la sesión, cuando la misma se crea desde el Demo (ver sección 3.5). En general, si fuera una actividad en el que hay interacción entre n participantes, el número que se debe considerar debe ser múltiplo de n. En el caso de una encuesta como "survey", se tiene por defecto $n = 1$, aunque podría ser cualquier otro número[13].

- ROOMS

[13]Que será, obviamente, múltiplo de uno.

ROOMS es una lista de diccionarios en donde se va a configurar las salas del experimento tales como el nombre de la sala (name/display_name) o las etiquetas de los jugadores (participant_label_file). Este último campo es esencial, pues se trabaja con etiquetas que guardan el anonimato de los participantes. Como podrá observar, el archivo de texto que contiene las etiquetas del room “Econ 101 class" se llama por defecto “_rooms/econ101.txt”, el cual está en la carpeta “_rooms”. Dichas etiquetas se pueden modificar según se requiera.



Fuente: Plantillas de oTree

- SESSION DEFAULTS

  En *SESSION_CONFIGS_DEFAULTS* usted podrá configurar algunas funcionalidades del aplicativo tales como la equivalencia entre moneda y puntos, y el fee del participante (pago) por estar en el experimento.

```
SESSION_CONFIG_DEFAULTS = dict(
    real_world_currency_per_point=1.00, participation_fee=0.00, doc=""
)

PARTICIPANT_FIELDS = []
SESSION_FIELDS = []

# ISO-639 code
# for example: de, fr, ja, ko, zh-hans
LANGUAGE_CODE = 'en'

# e.g. EUR, GBP, CNY, JPY
REAL_WORLD_CURRENCY_CODE = 'USD'
USE_POINTS = True
```

Fuente: Plantillas de oTree

- LANGUAGE CODE, REAL WORLD CURRENCY y USE POINTS

  En *LANGUAGE_CODES* podrá cambiar al idioma que desee, en este caso, está predeterminado el idioma inglés “en”. En *REAL_WORLD_CURRENCY_CODE*, se define la moneda con la que se quiera trabajar, por defecto es USD, pero puede usar PEN o SOLES, ya que es la moneda nacional. Por último, *USE_POINTS* toma el valor de “True” si el experimento trabaja con puntos, y el valor de “False” si trabaja con la moneda real.

Para conseguir familiaridad con estas configuraciones, se sugiere modificar algunas de ellas. Para guardar y visualizar los cambios realizados, presione "Control + S" en Visual Studio, y luego abra la sesión en el navegador (con el comando "otree devserver", tal y como se explica al inicio de la sección 3.5).

3.7. La carpeta del aplicativo

La carpeta de un aplicativo contiene, esencialmente, el archivo *__init.py__* y archivos tipo `html`[14]. El primero es el más importante, y contiene todas las variables, las funciones y define las páginas que tendrá el aplicativo. En la siguiente imagen se muestra la carpeta del aplicativo "survey", el cual incluye los archivos tipo html *CognitiveReflectionTest.html* y *Demographics.html*.

Fuente: Plantillas de oTree

Archivo __init__.py

El archivo *__init__.py* contiene lo siguiente: la clase "Constants", modelos de datos (subsesión, grupo y jugador) y las páginas (pages). Cabe mencionar que cada modelo de datos está definido por una clase al igual que las constantes, y los atributos de clases se llaman campos o variables.

- Constantes

  El primer grupo de variables (objetos) que se definen son las constantes. Estos son los parámetros que se mantienen fijos durante la sesión, y que no varían entre los jugadores. Las constantes pueden ser del tipo numbers, strings, booleans, lists, etc. Por ejemplo, la clase "Constants" tiene las siguientes variables: *NAME_IN_URL*, *PLAYERS_PER_GROUP* y *NUM_ROUNDS*. *NAME_IN_URL* es una constante del tipo string que permite renombrar el nombre del aplicativo en la URL. *PLAYERS_PER_GROUP* es una constante del tipo numérico que hace referencia al número de jugadores por grupo, es decir, al número de jugadores que participan en una actividad interactiva (juego). En la siguiente imagen se muestra las constantes del aplicativo "Survey". Dado que es una encuesta individual, no se forman grupos; por eso se escribe "None". Por último, *NUM_ROUNDS* es una constante numérica que indica la cantidad de rondas para el aplicativo, es

[14]Un archivo html es un lenguaje de código que se utiliza para estructurar y desplegar ciertos contenidos en una página web.

decir, cuántas veces deseamos que se repita esta[15].

```
class C(BaseConstants):
    NAME_IN_URL = 'survey'
    PLAYERS_PER_GROUP = None
    NUM_ROUNDS = 1
```

Fuente: Plantillas de oTree

- Modelo de datos (subsesión, grupo y jugador)

  El segundo grupo de variables que se definirán son los correspondientes a las clases de los modelos de datos: subsesión, grupo y jugador. Las variables de cada clase pueden ser de distinto tipo. En oTree se trabaja con campos de variables, entre los que podemos distinguir:

  - BooleanField (para valores verdadero / falso y sí / no).
  - CurrencyField (para importes de divisas).
  - IntegerField.
  - FloatField (para números reales).
  - StringField (para cadenas de texto).
  - LongStringField (para cadenas de texto largas; su widget de formulario es un área de texto de varias líneas).

  En la clase "Subsession", generalmente, se configuran aspectos relacionados a las rondas. La siguiente clase, "Group", nos permite especificar las variables concernientes a decisiones grupales en actividades interactivas[16]. En la siguiente figura, note que el aplicativo *Survey*, al referirse a una encuesta, no cuenta con variables en estas clases.

```
class Subsession(BaseSubsession):
    pass


class Group(BaseGroup):
    pass
```

Fuente: Plantillas de oTree

La última clase de este grupo, "Player", define variables a nivel individual (jugador). Como se ve en la siguiente imagen, en el aplicativo survey, la clase "Player"

[15] En una actividad no interactiva (como una encuesta), no tiene sentido la repetición; mientras que en una actividad interactiva (juego) sí.

[16] Como el juego de los bienes públicos que se presenta en el siguiente capítulo.

tiene cinco campos: *age*, *gender*, *crt_bat*, *crt_widget* y *crt_lake*. El campo *age* es del tipo entero y toma valores entre 13 y 125. El segundo campo, *gender*, el del tipo cadena de texto, y tiene dos opciones: Male y Female. Los tres últimos campos son del tipo entero, y tienen unas etiquetas (labels) que son las instrucciones. En la siguiente sección, observaremos estas configuraciones en el Demo.

```
class Player(BasePlayer):
    age = models.IntegerField(label='What is your age?', min=13, max=125)
    gender = models.StringField(
        choices=[['Male', 'Male'], ['Female', 'Female']],
        label='What is your gender?',
        widget=widgets.RadioSelect,
    )
    crt_bat = models.IntegerField(
        label='''
        A bat and a ball cost 22 dollars in total.
        The bat costs 20 dollars more than the ball.
        How many dollars does the ball cost?'''
    )
    crt_widget = models.IntegerField(
        label='''
        If it takes 5 machines 5 minutes to make 5 widgets,
        how many minutes would it take 100 machines to make 100 widgets?
        '''
    )
    crt_lake = models.IntegerField(
        label='''
        In a lake, there is a patch of lily pads.
        Every day, the patch doubles in size.
        If it takes 48 days for the patch to cover the entire lake,
        how many days would it take for the patch to cover half of the lake?
        '''
    )
```

Fuente: Plantillas de oTree

- Páginas (Pages)

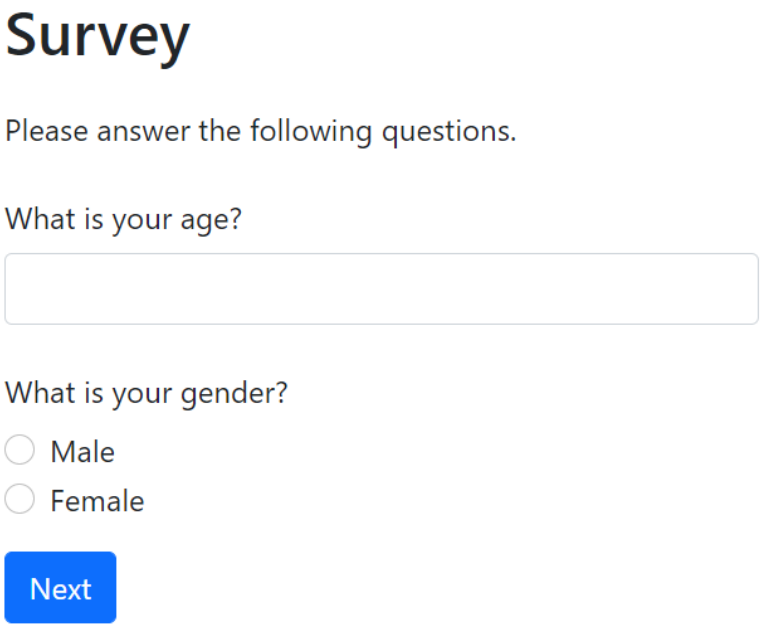
  Las páginas que observa el jugador en el Demo están definidas por la clase "Page". En este caso, en el aplicativo survey se han definido dos páginas: "Demographics" y "CognitiveReflectionTest". El nombre de las clases debe ser igual al nombre de los archivos `html` de la carpeta del aplicativo, tal como se observa en la imagen de abajo. Cada clase tiene dos campos: *form_model* y form_fields. El campo *form_model* es igual al nombre de la clase de donde se van a extraer los campos. Por ejemplo, puede ser *player* o *group*. Por otro lado, el campo *form_fields* es una lista que contiene los campos de la clase que se eligió en *form_model*.

  Finalmente, después de definir todas las clases para las páginas, es necesario precisar el orden en el que aparecerán las mismas. Esto se hace al final del archivo con *page_sequence*, la cual es una lista que indica el orden en el que se mostrarán las páginas al participante. En el aplicativo survey, primero aparecerán los cam-

pos de la clase “Demographics” en una página, y luego los campos de la clase “CognitiveReflectionTest” en la siguiente.

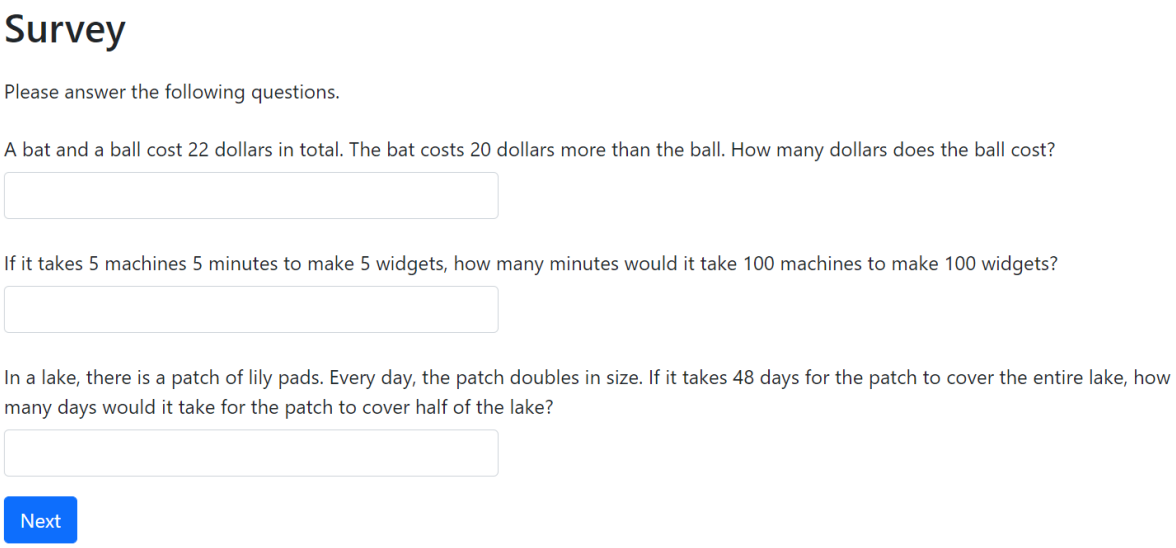
Fuente: Plantillas de oTree

En el aplicativo survey, la clase “Demographics” obtiene los campos *age* y *gender* de la clase “Player”. Esto significa que estos dos campos se solicitan en una página, tal como se puede observar en la siguiente imagen del Demo:



Fuente: Aplicativo en el servidor de oTree

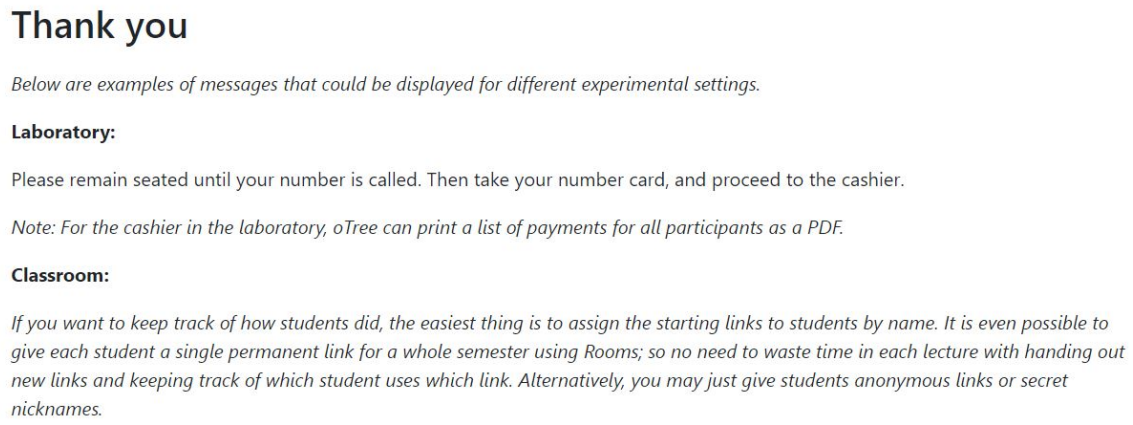
La siguiente página del aplicativo corresponde a la clase “CognitiveReflectionTest”. Aquí se solicitan los campos *crt_bat*, *crt_widget* y *crt_lake* de la clase “Player”. En el Demo se muestra la siguiente página:



Fuente: Aplicativo en el servidor de oTree

PaymentInfo

Por último, existe una página importante llamada *paymentInfo.html*, la cual no está en la carpeta del aplicativo de Survey, sino en el aplicativo llamado *payment_info*. Este archivo html nos permite ingresar la información de pago y resultados de nuestro experimento. Este aplicativo se usa, también, para redactar los comentarios finales y el agradecimiento a los participantes. Por su naturaleza, esta página debe mostrarse en todos los experimentos. Por ello, el aplicativo *payment_info* aparece al final de la secuencia de los aplicativos. El contenido, por defecto, es el siguiente:



Fuente: Aplicativo en el servidor de oTree

Archivos HTML

Como se señaló, además del archivo *__init.py__*, dentro de la carpeta del aplicativo *survey* podemos identificar archivos de tipo `html`. Estos se utilizan para desplegar páginas en la web.

Más formalmente, un archivo `html` es un lenguaje de código que se utiliza para estructurar y desplegar ciertos contenidos en una página web. En oTree ese contenido es propiamente el aplicativo creado, el cual es acompañado por el formato o diseño de esa página que se realiza con ciertos códigos propios de un archivo `html`.

Un template de oTree es una mezcla de dos lenguajes: `html`, que usa paréntesis angulares, como <this>... </this>); y *template tags*, que usa llaves {{ }}.

A continuación se explica más sobre los *template tags*. Para ver otras funciones que se pueden realizar con el lenguaje `html`, revisar el Apéndice A.4. En la siguiente imagen (Demographics.html) se puede observar que existen dos bloques:

- El título que empieza con {{`block title`}} y termina con un {{`endblock`}}.
- El cuerpo/contenido de la página está entre {{`block content`}} y {{`endclock`}}. Dentro de este bloque destacan tres etiquetas:
  - `formfields`: hace referencia a las variables que anteriormente se definieron en *__init__*.py.
  - `next_button`: con esta etiqueta se define el botón siguiente, el cual sirve para pasar de páginas, según el orden que se haya establecido en la configuración del aplicativo.
  - ... : este es un html tags y sirve para empezar a escribir un párrafo dentro del archivo `html`.

Dentro de las las llaves {{ }} se despliegan valores de variables previamente creadas; no se puede realizar operaciones aritméticas o modificar variables directamente. Asimismo, también se pueden desplegar las variables usando sentencias condicionales o en bucles de Python como `if` o `for`, respectivamente[17].

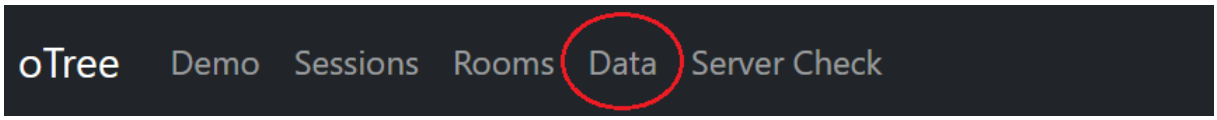
[17]Para más información: https://otree.readthedocs.io/en/latest/templates.html#step-1-otree-scans-template-tags-produces-html-a-k-a-server-side.

```
{{ block title }}Survey{{ endblock }}
{{ block content }}

    <p>
        Please answer the following questions.
    </p>

    {{ formfields }}

    {{ next_button }}

{{ endblock }}

```

Fuente: Plantillas de oTree

3.8. Data: descarga y descripción de variables relevantes

Esta sección describe en detalle cómo obtener/descargar los datos del resultado del experimento. Para empezar, acceda al local host de oTree como se explica en la sección 3.5. Estando en el Demo, luego de seleccionar *Survey*, aparece una barra superior negra con las pestañas Demo, Sessions, Rooms, Data y ServerCheck. Presione la opción Data.



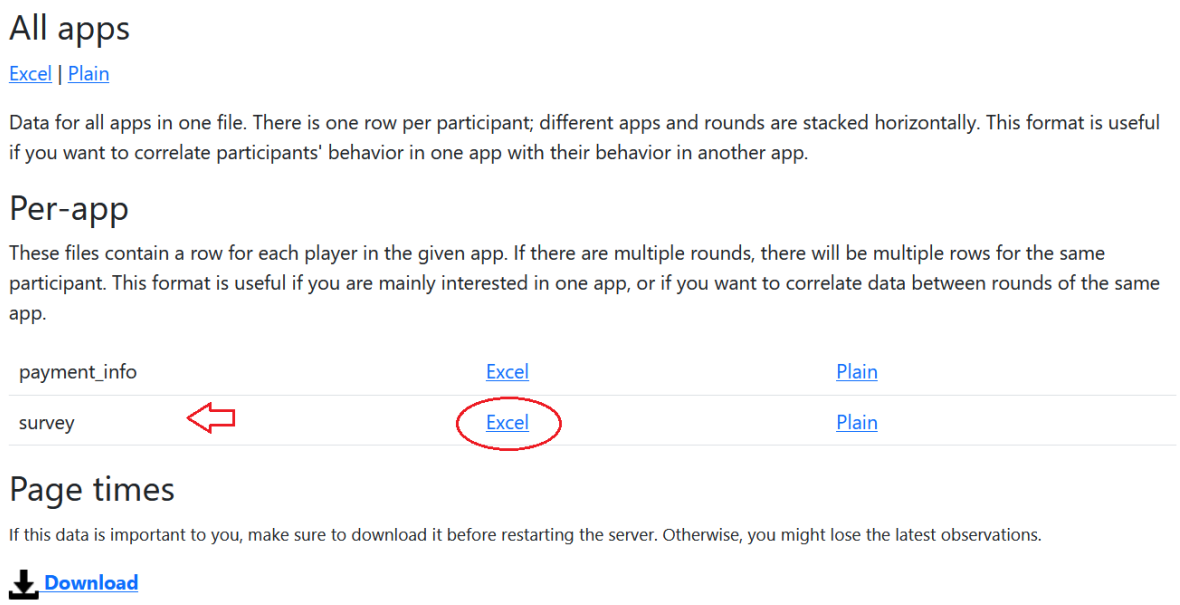
Fuente: Servidor local de oTree

Existen dos maneras de descargar los datos que se obtiene de un experimento: el archivo con la información de todas las aplicaciones o por aplicativo.

- Una primera forma de obtener los datos es descargando en un único archivo toda la información obtenida de todas los aplicativos de las diversas actividades que se hayan realizado en el Demo. Para ello, en el apartado "All apss" seleccione ´´Excel"[18].

[18]Ese archivo, a diferencia de los archivos por aplicativo, presenta una fila por participante, y los diferentes aplicativos y rondas se apilan horizontalmente

- Otra manera, en caso se requiera solo los datos que le proporciona un aplicativo en específico (en este caso *Survey*), es descargando, en el apartado "Per-app", los datos por cada aplicativo.



Fuente: Servidor local de oTree

Adicionalmente, se puede obtener un archivo del tiempo por página que demora cada jugador. En este caso, descargue el archivo del apartado “Page times". En ese archivo, cada fila contiene a cada jugador en la respectiva página que se encontraba en el aplicativo[19].

Para ver un ejemplo de descarga de datos, en esta oportunidad solo va a descargar el archivo csv que corresponde al aplicativo *Survey*, asumiendo que se tiene un participante que completó la encuesta. Dado que el archivo excel está en formato csv, es importante separar los datos por columnas. Para ello, siga los siguientes pasos:

1. Primero, seleccione toda la primera columna y diríjase a la opción Datos de la barra superior.

2. Segundo, en la casilla “Herramientas de datos” seleccione “Texto en columnas”.

[19]Realice la conversión del tiempo buscando en el navegador sobre el formato UNIX.

3. Finalmente, aparece un cuadro llamado "Asistente para convertir texto en columnas". En el primer paso, presione siguiente. En el segundo paso, seleccione en separadores "coma" u otro separador según como estén separados los datos en se archivo csv. En el tercer paso, seleccione finalizar.

Una vez que se puede visualizar la información por columnas, se procede a explorar los datos. Los archivos csv por aplicativo contienen una fila para cada jugador. Si hay varias rondas, habrá varias filas para el mismo participante. Este formato es útil si uno está interesado en una aplicación, o si deseas comparar datos entre rondas de una aplicación.

Las variables informativas se encuentran en toda la primera fila por columnas y se pueden identificar porque las variables empiezan con el nombre de la clase de la variable. En general, el orden es el siguiente: participant, player, group, subsession y session.

A continuación, se mencionarán las variables que usa este aplicativo:

- Primero, están las variables por la clase participant:
  - `participant.id_in_session`: Es una numeración de los participantes. El número de participantes se define dentro del diccionario que configura una sesión en el archivo *settings.py*. En este caso, dado que solo hay un participante predeterminado, el id es 1. Sin embargo, si cambiamos el número de participantes a 9 (por ejemplo) y completa las 9 encuestas, en los datos descargados se podrá notar un numeración del 1 al 9 en esa columna.
  - `participant.code`: Presenta un código específico que el servidor por defecto asigna a cada participante.
  - `participant.label`: Aparecen las etiquetas que el experimentador asigna a cada participante, con el fin de mantener la confidencialidad.
- Segundo, las variables de la clase player (ver la siguiente imagen):
  - `player.id_in_group`: Es una numeración de los jugadores en un grupo. En el caso de una encuesta, donde existen grupos de un jugador, la numeración es similar a la de `participant.id_in_session`.
  - `player.age y player.gender`: Estas son variables específicas del aplicativo que se crearon para la encuesta y representan variables demográficas.

- `player.crt_bat, player.crt_widget y player.crt_lake`: Estas también son variables definidas para el aplicativo que el encuestado respondió.

Q	R	S	T	U
player.age	player.gender	player.crt_bat	player.crt_widget	player.crt_lake
23	male	34	56	60

- Tercero, las variables de la clase group: en este caso no se va mostrar ninguna variable relevante para el aplicativo, ya que no existen variables de grupo creadas.
- Cuarto, las variables de la clase subsession:
  - `subsession.round_number`: Enumera a las rondas que se realizaron. En este caso, solo hay una ronda.
- Quinto, las variables de la clase session:
  - `session.code`: Es el código de la sesión que crea por defecto el servidor oTree.

3.9. Ejercicios propuestos

1. Encuesta demográfica

   Siguiendo la plantilla del aplicativo *survey*, que se describió en este capítulo, elabore una encuesta. Este ejercicio consiste en elaborar tres páginas:

   a) En la primera página, se le debe preguntar la edad al participante, la cual tiene que estar en el rango de 18 a 100 años.

   b) En la segunda página, se le debe preguntar por su carrera, ciclo de estudios y especialidad. Respecto al ciclo de estudios, las opciones deben mostrarse en una lista desplegable.

   c) En la tercera página (*payment info*), el participante debe visualizar lo siguiente:

   - La información de pago reescrita en español.
   - Una tabla que tenga como encabezados: **Juegos de Economía** y **Número de participantes por grupo**. Para completar los datos de la tabla, elija 4 aplicativos de los que aparecen en la carpeta de su proyecto.
   - Una imagen del logo de su universidad.

   Notas:

   - Toda la encuesta debe estar en español. Transcribir los apartados de inglés a español.
   - Ver los siguientes links que le ayudarán en este ejercicio:

- Lista desplegale:

  https://otree.readthedocs.io/en/latest/forms.html?highlight=choices

- Tabla:

  https://www.tablesgenerator.com/html_tables

- Imagen:

  https://otree.readthedocs.io/en/latest/misc/advanced.html#staticfiles

2. Encuesta de Kahneman y Tversky

   Realice una encuesta basada en la de Kahneman y Tversky (1979) (ver páginas 265-267), el cual estudia las decisiones de los individuos bajo condiciones de riesgo. La encuesta debe incluir cuatro preguntas de los ocho problemas del documento.

   Pista: Le recomendamos buscar en la documentación de oTree cómo utilizar `widget.RadioSelect` para que las opciones de la encuesta sean botones.

   Mayor información: https://otree.readthedocs.io/en/latest/forms.html#forms

Capítulo 4

Programación de Experimentos II

4.1. Introducción

En este capítulo se introducen nuevos aspectos de la programación de experimentos en oTree, los cuales permiten programar actividades con interacción entre los participantes (juegos). Dentro de estos aspectos, se tiene el uso de la clase Group, y la elaboración de funciones. Naturalmente, estas interacciones dan lugar a unos datos experimentales mucho más complejos, los cuales también exploramos en este capítulo. Para ilustrar estos nuevos aspectos, se considera el juego de bienes públicos, el cual viene predeterminado en oTree, y que ya se encuentra instalado en la carpeta de *primer_proyecto*.

El juego de bienes públicos consiste en analizar el comportamiento del *free rider* mediante el mecanismo de contribuciones voluntarias. En este juego, cada individuo posee una dotación inicial, y a partir de esa decide qué parte va a contribuir voluntariamente al grupo. Estas contribuciones individuales se suman, y el total se multiplica por un multiplicador (a elección del experimentador). Este nuevo total se divide en partes iguales entre los jugadores del grupo. De esta manera se obtiene la porción individual de cada jugador. Es decir,

$$\text{Porción individual} = \frac{\text{Suma Contribuciones} * \text{Multiplicador}}{\text{Número de participantes}}$$

El pago (*payoff*) de cada jugador del grupo se define de la siguiente forma:

$$\text{Payoff} = \text{Dotación} - \text{Contribución} + \text{Porción Individual}$$

Es importante entender la lógica detrás de estas fórmulas, porque, más adelante, las veremos en el código del juego. Aunque se asume que ha leído el capítulo anterior, por razones pedagógicas, algunos elementos básicos vistos previamente, se repite en este capítulo.

Este capítulo contiene siete secciones. En la presente sección se ha descrito el objetivo del capítulo, y el juego de bienes públicos que servirá como ilustración. En la sección 2 se revisa la configuración general de un proyecto, mostrando cómo hacer que un aplicativo aparezca en el Demo del servidor local de oTree. En la sección 3 se trata de la

carpeta del aplicativo: se aprende a crear variables a nivel de la clase Group y a crear y utilizar funciones. Asimismo, se trata sobre cómo mostrar información en forma dinámica mediante los template tags. En la sección 4 se explora la data de un aplicativo en el cual hay interacción entre los participantes. La sección 5 contiene ejercicios propuestos de nivel básico. La sección 6 contiene un ejercicio de nivel avanzado. Para profundizar en la programación de experimentos, la última sección sugiere programar algún experimento descrito en algún artículo académico.

4.2. Configuración general del proyecto

Como se señaló en el capítulo anterior, para que un aplicativo aparezca en el *demo* del servidor local, primero se necesita crear un diccionario como elemento de la lista *SESSION_CONFIGS* en el archivo *settings.py*. En este nuevo diccionario se agrega el nombre del juego que quiere que aparezca en el demo, en este caso, "Bienes Públicos". Asimismo, es importante tener en cuenta la secuencia de los aplicativos: primero el aplicativo del juego propiamente dicho, y después la información de pago. También se debe precisar el número de participantes de la sesión. El código es el siguiente:

```
dict(
    name='BienesPublicos', display_name="Bienes Públicos",
    app_sequence=['public_goods_simple', 'payment_info'], num_demo_participants=3
),
```

Fuente: Plantillas de oTree

Finalmente, el código completo de *SESSION_CONFIGS* se visualiza de la siguiente manera:

```
SESSION_CONFIGS = [
    dict(
        name='guess_two_thirds',
        display_name="Guess 2/3 of the Average",
        app_sequence=['guess_two_thirds', 'payment_info'],
        num_demo_participants=3,
    ),
    dict(
        name='survey', app_sequence=['survey', 'payment_info'], num_demo_participants=1
    ),
     dict(
        name='BienesPublicos', display_name="Bienes Públicos",app_sequence=['public_goods_simple', 'payment_info'], num_demo_participants=3
    ),
]
```

Fuente: Plantillas de oTree

Por el capítulo previo, se sabe que otros elementos importantes dentro de *SESSION_CONFIGS* son la configuración del idioma y de los puntos del juego. Para verificar que el idioma es Español, al lado de *LANGUAGE_CODE* tiene que estar 'es', que viene a configurar el idioma español. En caso se quiera cambiar a otros idiomas, solo es necesario cambiar el 'es' por otro código de idioma como; por ejemplo, 'en'

(inglés), 'fr' (francés) o cualquier otro código de idioma dentro del ISO-639.

Por otro lado, para configurar que los puntos del juego aparezcan con una moneda local, se necesita poner, al lado de *REAL_WORLD_CURRENCY_CODE*, el código de la moneda de preferencia, en este caso, "PEN" (Nuevos soles). En caso se requiera cambiar a dólares u otra moneda, solo es necesario poner el código correspondiente a la moneda. Asimismo, para que aparezca como unidad de puntos la moneda local seleccionada, es importante verificar que al lado de *USE_POINTS* se encuentre el comando **False**.

En la siguiente imagen se puede ver la configuración de estos aspectos:

```
# ISO-639 code
# for example: de, fr, ja, ko, zh-hans
LANGUAGE_CODE = 'en'

# e.g. EUR, GBP, CNY, JPY
REAL_WORLD_CURRENCY_CODE = 'PEN'
USE_POINTS = False
```

Fuente: Plantillas de oTree

4.3. La carpeta del aplicativo

El objetivo de esta sección es aprender a programar en oTree actividades en las que los participantes interactúan. Para ilustrar la forma de hacer esto, se presentan los componentes del juego de bienes públicos, el cual se describe en la carpeta *public_goods_simple*.

Como explicamos en el capítulo anterior, la carpeta de un aplicativo contiene el archivo *__init.py__* y archivos tipo html. Dado que, a diferencia de una encuesta, en el juego de bienes públicos hay interacción entre los participantes de un grupo, es necesario definir variables a nivel de la clase Group, y una función en el archivo *__init.py__*.

Por completitud, se describe en detalle el contenido de la carpeta del aplicativo de bienes públicos, revisando, de paso, algunos elementos ya estudiados en el capítulo previo.

- Constantes

  En este aplicativo de bienes públicos, la clase "Constants" tiene cinco campos: *NAME_IN_URL*, *PLAYERS_PER_GROUP*, NUM_ROUNDS, *ENDOWMENT* y *MULTIPLIER*. El primero, es para especificar el nombre del URL del juego. El segundo, es para especificar el número de jugadores por grupo; en este caso, se forman grupos de 3. El tercero, es para especificar el número de rondas del juego; por defecto, solo es una ronda, pero para fines didácticos, se coloca 2 rondas. El cuarto, es para especificar la dotación de cada jugador. Aquí, `cu()` hace referencia

al diminutivo del término currency, y denota a las divisas. Finalmente, el quinto, es para especificar el multiplicador de la contribución total del grupo.

Cabe resaltar que, a diferencia del aplicativo *Survey*, *PLAYERS_PER_GROUP* es tres y no uno; precisamente, para indicar que los tres participantes interactúan, en el sentido de que los pagos individuales dependen de las decisiones que tome el propio jugador y las de los otros dos.

```
class C(BaseConstants):
    NAME_IN_URL = 'public_goods_simple'
    PLAYERS_PER_GROUP = 3
    NUM_ROUNDS = 2
    ENDOWMENT = cu(100)
    MULTIPLIER = 1.8
```

Fuente: Plantillas de oTree

- Modelo de datos (subsesión, grupo y jugador)

  Los modelos de datos, como se vio en el capítulo 3, son los siguientes: Subsession, Group y Player. La clase "Subsession" sirve para configurar aspectos relacionados a las rondas. Si no hay alguna configuración por hacer en una clase, se coloca `Pass` como se observa en la imagen de abajo. La siguiente clase, "Group", nos permite especificar las variables concernientes a decisiones grupales. Por ejemplo, en el caso del juego de bienes públicos, tendremos decisiones donde hay presencia de una interacción grupal, por lo cual se hará uso de esta clase. Aquí, están definidos dos campos: *total_contribution* e *individual_share*. El primero es del tipo divisa, y corresponde a la contribución total de los jugadores de un grupo. El segundo representa la porción de la contribución total del grupo que le corresponde a cada jugador del grupo. Por último, la clase "Player" contiene un único campo, *contribution*, el cual se refiere a la contribución individual de cada jugador al grupo.

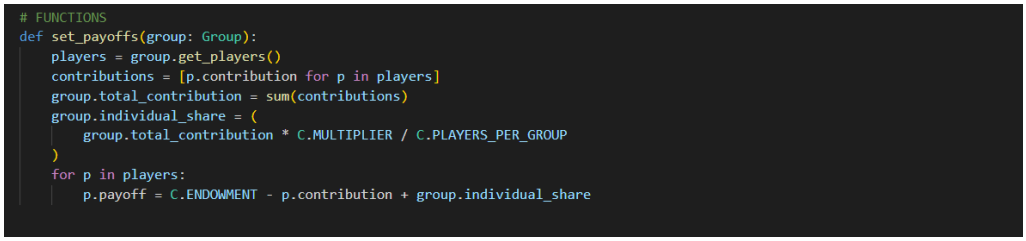
  Cabe enfatizar que, a diferencia del aplicativo *Survey*, ahora sí es necesario usar la clase Group. Asimismo, note que los campos *total_contribution* e *individual_share* no pueden ser definidos como constantes (clase C), pues, si bien estas variables son constantes en cada grupo, el valor de las mismas podría variar de un grupo a otro.

```
class Subsession(BaseSubsession):
    pass


class Group(BaseGroup):
    total_contribution = models.CurrencyField()
    individual_share = models.CurrencyField()


class Player(BasePlayer):
    contribution = models.CurrencyField(
        min=0, max=C.ENDOWMENT, label="How much will you contribute?"
    )
```

Fuente: Plantillas de oTree

- Funciones

  Después de haber definido los campos de las clases anteriores, y si el experimento lo amerita, se continúa definiendo las funciones. Estas son estructuras que permiten crear la interacción que se realizará dentro del juego, ya sea a nivel de subsesión, grupo o jugador. En el caso del aplicativo de bienes públicos, existe una función que define los pagos correspondientes a cada jugador, dependiendo de su contribución y de la contribución grupal. De esta manera, se llama a las variables definidas previamente y se construye la fórmula para el cálculo de los pagos, la cual es la siguiente:



```python
# FUNCTIONS
def set_payoffs(group: Group):
    players = group.get_players()
    contributions = [p.contribution for p in players]
    group.total_contribution = sum(contributions)
    group.individual_share = (
        group.total_contribution * C.MULTIPLIER / C.PLAYERS_PER_GROUP
    )
    for p in players:
        p.payoff = C.ENDOWMENT - p.contribution + group.individual_share
```

Fuente: Plantillas de oTree

En el caso de bienes públicos, se tiene la función *set_payoffs*, la cual se ha creado a nivel de la clase Group. Esto es importante, pues se utilizan las variables *total_contribution* e *individual_share*, creadas a nivel de esta clase.

A continuación presentamos la explicación del código de la imagen previa. Primero, se tiene la variable *players*, la cual es una lista de los jugadores por grupo identificados por un id. Para lograrlo, se hace uso de *get_players*, el cual es un método de la clase Group que ya viene creada en oTree. Segundo, la variable *contributions* es una lista que contiene las contribuciones de cada jugador. Tercero, *group.total_contribution* es la suma de los elementos de la lista anterior. Cuarto, *group.individual_share* es la porción de la contribución total que le corresponde a cada jugador. Esta se define como la contribución total multiplicado por la constante *MULTIPLIER* y dividido por la constante *PLAYERS_PER_GROUP*. Por último, con un loop se calcula el pago para cada jugador. Este es la resta de la constante *ENDOWMENT* menos la contribución del jugador (*p.contribution*) sumado a la variable *group.individual_share*. Cabe notar que en este último paso se hace uso de *payoff*, el cual es un campo de la clase Player, y que ya viene predeterminado en oTree.

- Páginas

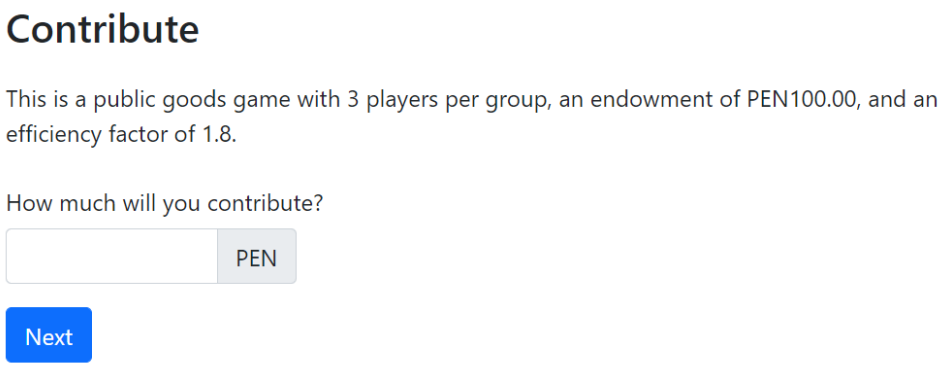
  En esta sección del archivo haremos algunas configuraciones para la visualización del juego en la página web. Por ejemplo, como ya se mencionó en el capítulo 3, *page_sequence* es una lista acerca del orden en el que se mostrarán las páginas al participante. En el juego de bienes públicos, por defecto, primero aparece la página de contribución, luego la página de espera hasta que los otros jugadores decidan y, finalmente, en la última página aparecen los resultados.

Esta parte del script trabaja con los archivos html *Contribute* y *Results*. Tener en cuenta que al definir una clase de página el nombre debe ser el mismo que el del archivo `html`, sino no lo reconoce. Veamos cada clase para comprender mejor esta interacción.

Fuente: Plantillas de oTree

- Contribute

  "Contribute" es una instancia de la clase Page. Esta instancia trabaja con el archivo `html` del mismo nombre. Si no definimos la clase "Contribute" en el script (y, por lo tanto, no la llamamos en *page_sequence*), el participante no podrá visualizar la ventana configurada en "Contribute.html". Esta página contiene los campos *form_model* y *form_fields*, los cuales corresponden a la clase "Player"[1]. En el Demo del local host, la página se ve de la siguiente manera:



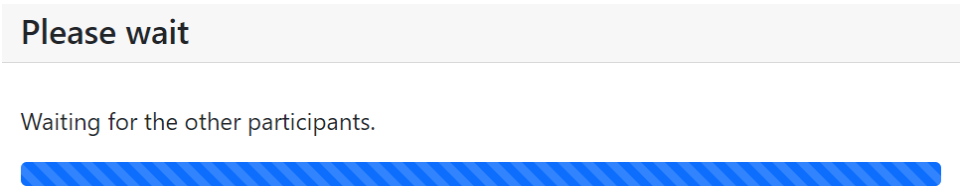
Fuente: Aplicativo en el servidor de oTree

En la imagen anterior, "efficiency factor" se refiere al Multiplicador.

[1]En el caso que omita *form_fields*, se abre la ventana de decisión, pero no se permite llenar la cantidad solicitada.

- ResultsWaitPage

  "ResultsWaitPage" es una instancia de la clase WaitPage, la cual generará un mensaje de espera, luego de que el jugador realice su contribución correspondiente. Al poner que el campo *after_all_payers_arrive* es igual a la función *set_payoffs*, definida anteriormente, se calcula el pago final para cada jugador. Por eso, es necesario esperar a que todos los jugadores coloquen su contribución. En el Demo del local host se observaría lo siguiente:

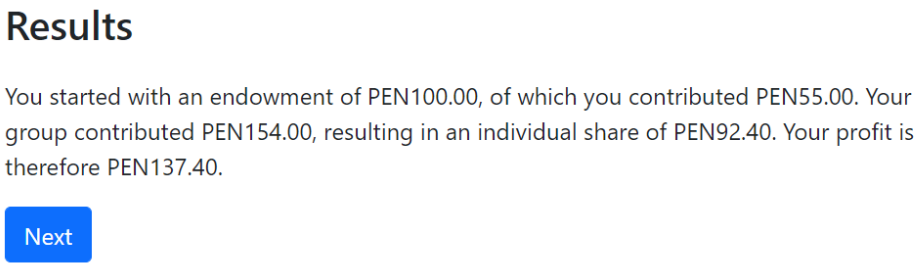


Fuente: Aplicativo en el servidor de oTree

  Es importante enfatizar que la página "ResultsWaitPage" se requiere en toda actividad que implique interacción de los participantes.

- Results

  "Results" es otra instancia de la clase Page. Esta instancia trabajará con el archivo *Results.html*. Debido a que solo se mostrará el contenido de dicho archivo, entonces se coloca `pass`. En el caso de bienes públicos, la página de un participante se vería de la siguiente manera:



Fuente: Aplicativo en el servidor de oTree

Por último, como se señala en el capítulo anterior, existe una página importante llamada *paymentInfo.html*, la cual está en el aplicativo llamado *payment_info*. Este aplicativo es el último que aparece en la secuencia de aplicativos, pues esta página se muestra al terminar el experimento.

Archivos HTML

A continuación estudiamos en detalle el contenido de las páginas del aplicativo de bienes públicos. Esta parte complementa lo que se trató sobre archivos html en el capítulo

3.

Un aspecto nuevo que se encuentra en las páginas del aplicativo es el uso de los *template tags*, lo cual sirve para mostrar información de forma dinámica. El procedimiento es el siguiente. La variable que se llama se pone entre llaves. Para llamar a la variable se coloca el nombre de la clase, seguido de un punto y el nombre de la variable que se quiere llamar. Por ejemplo, para llamar a la variable de dotación ("ENDOWMENT") se escribe {{C.ENDOWMENT}}. El archivo *Contribute.html* se visualiza así:

```
[ ]: This is a public goods game with
     {{ C.PLAYERS_PER_GROUP }} players per group,
     an endowment of {{ C.ENDOWMENT }},
     and an efficiency factor of {{ C.MULTIPLIER }}.
```

Ciertamente, en general, se pueden llamar las variables creadas en las clases: session, subsession, participant, group, player, y C (de constantes). Las variables de la clase Player permiten que en la página se muestre lo que corresponde al jugador que está viendo la página en el momento; las variables de la clase Group, muestran lo que describe la variable en el grupo al cual el jugador pertenece.

Por ejemplo, en el archivo *Results.html* se informa al jugador, entre otros datos, sobre su pago por ronda, escribiendo {{ player.payoff }}. Aquí reproducimos el código de esta página:

```
[ ]: You started with an endowment of {{ C.ENDOWMENT }},
     of which you contributed {{ player.contribution }}.
     Your group contributed {{ group.total_contribution }},
     resulting in an individual share of {{ group.individual_share }}.
     Your profit is therefore {{ player.payoff }}.
```

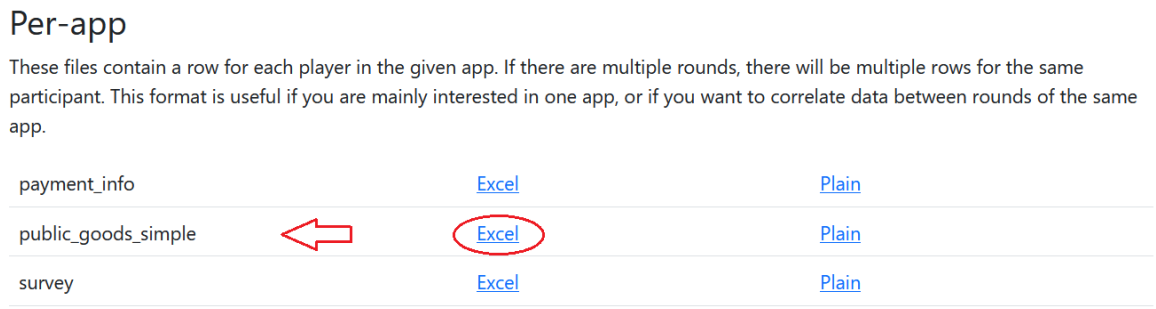
Cabe indicar que en archivos tipo html se puede incluir imágenes, tablas y texto en general. Para ver cómo realizar estas funcionalidades y más, ver el Apéndice A.4.

4.4. Data: descarga y descripción de variables relevantes

En este sección se describen algunas variables de los datos que no se mencionaron en la sección "Data' del anterior capítulo, utilizando ahora como ejemplo el juego interactivo de bienes públicos. De esta manera, ambas secciones se complementan.

Para empezar, ejecute el juego de bienes públicos con tres participantes y dos rondas. Complete los datos que se solicitan en ambas rondas.

Luego de completar la ejecución, descargue, en el apartado "Per-app", el archivo csv específico del aplicativo *public_goods_simple*.



Fuente: Servidor local de oTree

Ahora, separe por columnas los datos usando los pasos que ya se describieron para el aplicativo *Survey* en el capítulo 3.

Luego de que ya se puede visualizar la información por columnas, se procede a explorar los datos. Como ya se mencionó, los archivos csv por aplicativo contienen una fila para cada jugador. Si hay varias rondas, habrá varias filas para el mismo participante.

Recordar que el orden se puede identificar porque las variables empiezan con el nombre de la clase de la variable: participant, player, group, subsession, y session.

A continuación, solo haremos mención de algunas variables de la clase participant, player, group y subsession:

- Participant:
  - `participant.id_in_session`: Es una numeración de los participantes. En este caso, se puede observar que son 3 participantes, con id 1, 2 y 3. Estos números aparecen dos veces porque son dos rondas.
  - `participant.code`: Presenta un código específico que el servidor por defecto asigna a cada participante. En este caso, el código se repite dos veces porque cada participante jugo en dos rondas.
  - `participant.payoff`: Se listan los pagos acumulados que obtuvo cada participante en las distintas rondas. Por ejemplo, el participante 1 tiene 245, pues obtuvo un pago de 145 en la ronda 1, y 100 en la ronda 2.

participant.code	participant.payoff	player.payoff	subsession.round_number
1sbci9xe	245	145	1
j56lmi1w	234	134	1
4bu6l9gj	212	112	1
1sbci9xe	245	100	2
j56lmi1w	234	100	2
4bu6l9gj	212	100	2

- Player:
  - `player.payoff`: A diferencia del `participant.payoff`, este da el pago de cada una de las rondas (no la acumulación).
  - `player.contribution` Esta es una variable específica del juego, y representa el monto que cada jugador contribuye al grupo.
- Group:
  - `group.total_contribution`: Es una variable que es programada en la clase Group. Representa la suma de las contribuciones de los 3 integrantes de un grupo en cada ronda.
  - `group.individual_share`: Está variable también es programada en la clase Group. Representa la cuota individual que obtiene cada miembro del grupo en cada ronda.
- Subsession:
  - `subsession.round_number`: Enumera a las rondas que se realizaron. En este caso, puede ver que se programaron dos rondas.

4.5. Ejercicios propuestos

1. Juego de bienes públicos con sanción

   Modifique el juego original de bienes públicos (plantilla predeterminada de oTree) de la siguiente manera:

   a) En el archivo settings.py, se debe de cambiar el formato de la moneda de USD a PEN. Que se vea la moneda en vez de los puntos.

   b) En el archivo _init_.py, se debe de especificar 4 participantes por grupo y dos rondas.

   c) La dotación inicial debe ser 90 soles y la constante *multiplier* debe ser 2.

   d) Agregar una variable constante de tipo *currency* llamada sanción. Esa variable va ser agregada como una sanción de 5 soles al pago final del participante si la contribución de este es menor a 20 soles.

   e) El texto del título y del contenido de las páginas deben estar en español. En el archivo html *Contribute* debe agregar que habrá una sanción si la contribución individual es menor a 20 soles. Mostrar el monto de esa sanción utilizando template tags.

   f) En el archivo html *Results* como parte de los resultados se debe mencionar si el participante fue sancionado o no, para eso debe utilizar template tags de tipo *conditions if*.

   g) En la página de información de pago agregue el logo PUCP.

   Enlaces de ayuda

- Ver "Conditions if" para poner el texto de sanción en el archivo "Contribution": https://otree.readthedocs.io/en/latest/templates.html

2. Reto: tiempo límite en el juego de bienes públicos

   En el ejercicio anterior, agregue un contador de 60 segundos por cada vez en la que el participante decida su contribución.

Enlace de ayuda

- Ver "timeout_seconds" en: https://otree.readthedocs.io/en/latest/timeouts.html?highlight=60#timeout-that-doesn-t-submit-the-page

4.6. Ejercicio avanzado: experimento de pesca

Este ejercicio consiste en replicar una versión simplificada del proyecto de investigación "Estrategias de Gestión de la Pesca Artesanal en Zonas Marino Costeras de Perú: Evidencia de un Experimento de Campo", a cargo de Carlos Soncco, estudiante de doctorado en economía en la PUCP. Esta investigación estudia el comportamiento de los agentes económicos durante la toma de decisiones respecto al uso y manejo de los recursos pesqueros, los cuales tienen la característica de ser recursos de uso común.

Aunque el trabajo original considera varios escenarios, en este ejercicio se replica el código del experimento solo para dos: sin regulación (grupo de control), y con regulación que implica una sanción monetaria (grupo de tratamiento). Cabe indicar que se tiene diez rondas, donde cada una representa un día de trabajo, y los participantes forman grupos de 4 personas. Para este ejercicio deberá trabajar con una plantilla del GitHub de LEEX-PUCP[2].

Primer escenario (Grupo Control):

En esta fase no hay regulación. Los participantes deberán elegir extraer el recurso pesquero entre 1 y 10 unidades. El pago final de cada jugador depende de su decisión y también del colectivo, ya que mientras mayor sea la extracción grupal, menor es el valor de cada unidad extraída para cada jugador. El pago final del individuo i se define con la siguiente ecuación:

$$\pi_i = 8g_i - 0{,}3g_i^2 + 80 - 2(g_1 + g_2 + g_3 + g_4)\,, \quad i = 1,2,3,4 \tag{4.1}$$

Donde g_i denota las unidades extraídas del participante i.

En la plantilla de la carpeta *public_goods_simple* se le pide completar lo siguiente:

- En la clase "Constants", defina las variables "players_per_group" y "num_rounds". Estas determinan el número de jugadores por grupo y la cantidad de rondas del juego respectivamente.

[2] https://github.com/leexpucp/Plantilla-EjercicioAvanzado

- En la clase "Player", defina la variable "contribution". Esta se refiere a la extracción del recurso pesquero por cada jugador. Recordar que se debe encontrar en el rango de 1 a 10 unidades. Asimismo, no olvidar agregar un label.
- Definir la función llamada "set_payoff", la cual determina el pago de cada jugador de acuerdo a la ecuación 4.1. Para esto, se debe guiar de la función "set_payoff" del juego de bienes públicos[3].
- Agregar un contador de 60 segundos en la página donde el participante decide la cantidad del recurso a extraer. Asimismo, alertarle cuando falten 30 segundos[4]. La alerta se debe programar en el archivo html "Contribute".

Segundo escenario (Grupo Tratamiento):

En esta fase hay una regulación externa, la cual consiste en una sanción monetaria de 9.6 por unidad adicional para las personas que extraigan más de una unidad del recurso pesquero. Sin embargo, el monitoreo es imperfecto, ya que la probabilidad de que este se realice es de 1/2 por cada grupo. Si se realiza el monitoreo a un grupo, se elige al azar a un integrante del mismo para que sea supervisado y se verifique su nivel de extracción. Para esto, todos los miembros de un grupo tienen la misma probabilidad de ser supervisados. Si un participante fue supervisado y extrajo más de una unidad del recurso pesquero, se le impondrá la sanción, y su pago final disminuye. Cabe mencionar que los participantes toman su decisión de manera privada y confidencial, es decir, no hay comunicación entre los jugadores, y nadie conoce el resultado de la inspección.

En la plantilla de la carpeta "public_goods_simple_sancion" se le pide completar lo mismo que en la anterior, pero agregar lo siguiente:

- En la clase "Constants", defina la constante sanción igual a 9.6.
- En la clase "Group", defina las variables "random_num" y "random_player_id". Las dos primeras tienen un valor inicial de cero. Para ello, siga el ejemplo de las variables ya definidas en la clase Group llamadas "total_contribution" y "other_contribution".

  Nota: La variable "group.random_num" se empleará para saber si hubo o no monitoreo del grupo en la ronda, y "group.random_player_id" hace referencia al número aleatorio del participante dentro del grupo, que será útil para saber si el participante fue supervisado.
- En la clase "Player", defina las variables "sancion_monto", "pago_final", "supervision" y "monitoreo".

  Las dos primeras variables hacen referencia al monto total de la sanción para cada participante y al pago final descontado la sanción, respectivamente. Ambas variables deben tener un valor inicial de cero.

  Por otra parte, la variable "supervision" es del tipo string con un valor inicial de "no supervisado". Finalmente, la variable "monitoreo" es del tipo string con un valor inicial de "no hay monitoreo".

[3] https://otree.readthedocs.io/en/latest/tutorial/part2.html?highlight=set_payoff
[4] https://otree.readthedocs.io/en/latest/timeouts.html

- En la función “set_payoffs”, adicional al código del primer escenario, definir lo siguiente:
  - Los valores de las variables “group.random_player_id” y “group.random_num”, donde la primera toma valores aleatorios enteros en un rango de 1 a 4 y la segunda 0 o 1. [5].
  - Dentro de la sentencia `for` que se utiliza para determinar el pago, creará dos sentencias condicionales (solo con `if`).

    En la primera utilice la variable grupal definida antes “group.random_num” y la variable de jugador “p.monitoreo”. La sentencia debe seguir la siguiente lógica: Si el número aleatorio es menor o igual a 0.5, entonces “sí hay monitoreo”.

    En la segunda utilice la variable de jugador “p.id_in_group” [6], y las variables definidas anteriormente “p.supervisión” y “group.random_player_id”. La sentencia debe seguir la siguiente lógica: Si el id del jugador es igual al numero aleatorio obtenido por “group.random_player_id”, que toma valores de 1 a 4, entonces la variable “p.supervision” debe ser igual a “sí fue supervisado”.
  - Por último, para determinar el pago final en donde se incluya la sanción determinada debe seguir la siguiente lógica en una sentencia condicional (`if / else`):

    Si el grupo fue monitoreado y el jugador fue supervisado y su contribución fue mayor a 1, primero definir el monto de la sanción como la sanción (la variable constante) multiplicada por la contribución del jugador menos 1. Luego, determinar el pago final como el pago menos el monto de la sanción. En caso contrario, el pago final es igual al pago sin ninguna sanción.
- En la página “Pagos”, coloque un mensaje similar al siguiente:

  “En su grupo {{sí (no) hay monitoreo}} y {{sí (no) fue supervisado}}[7] . Su sanción fue {{cantidad}}. Tu puntaje sería {{cantidad}}, pero por la sanción es {{cantidad}}”.

4.7. Fuente de otros ejercicios avanzados

Para profundizar sus habilidades de programación, un trabajo interesante puede ser programar un experimento publicado en algún paper. Dicho paper debe contener una explicación detallada del diseño del experimento. Asimismo, es preferible que el experimento haya sido ejecutado utilizando oTree. Puede encontrar experimentos en los siguientes journals:

[5] Puede utilizar la función random.randit para ambas variables

[6] Es una variable predeterminada por oTree que indica el identificador de cada jugador por grupo. No es necesario que se defina en la clase “Player”.

[7] Para ello haga uso de las variables creadas en el código “player.monitoreo” y “player.supervision”

- Journal of Behavioral and Experimental Economics
- Journal of Behavioral and Experimental Finance

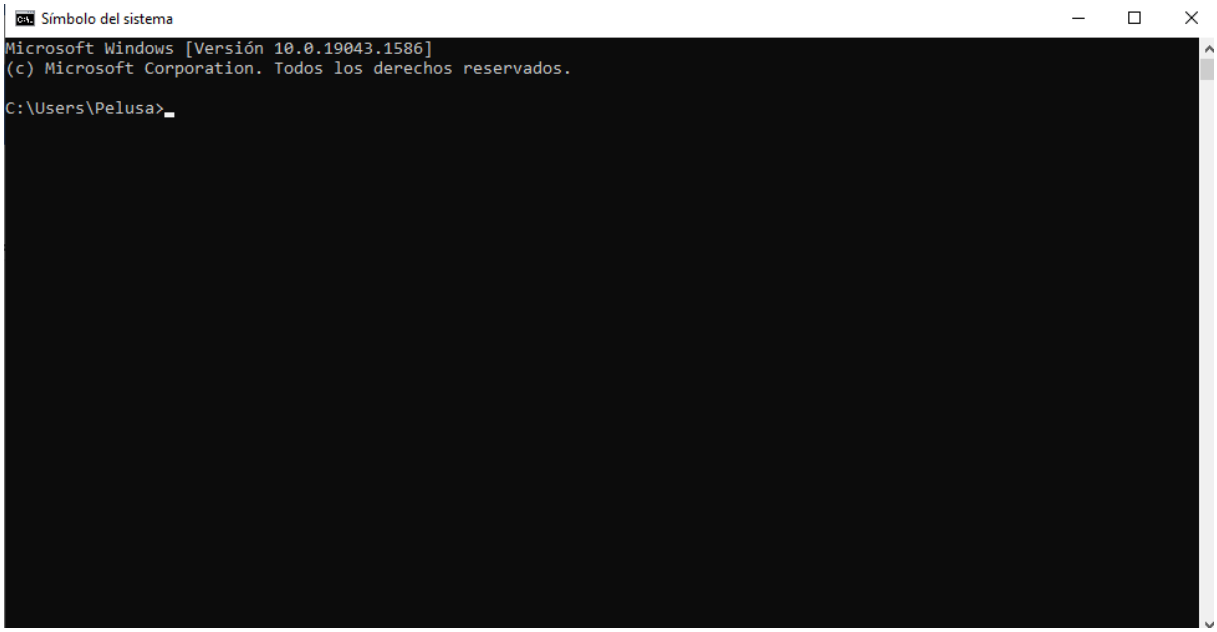
Apéndice A

Complementos

Aunque todas las aplicaciones o sus equivalentes se pueden usar en diferentes sistemas operativos, en este documento se usa Windows.

A.1. Símbolo del sistema o CMD

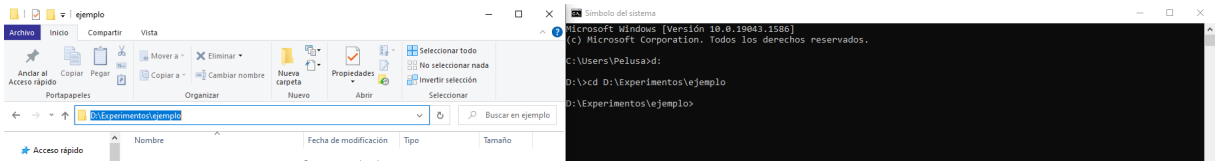
El símbolo de sistema de Windows o CMD es un intérprete de línea de comandos, los cuales son introducidos mediante la sintaxis de líneas de texto y ejecutados con la tecla enter. El CMD tiene diversas funciones como acceder/crear carpetas, eliminar archivos, acceder a páginas web, entre otros. Para abrirlo, se escribe en el buscador del escritorio "cmd" y, luego se selecciona "Símbolo del sistema". Aparece la siguiente pantalla:



Como se observa, el símbolo del sistema se encuentra en la unidad del usuario, que en

este caso es C. Para cambiar de unidad, por ejemplo, D, se escribe simplemente “d:”. A continuación, algunos de los comandos básicos del CMD:

- **cd**: permite cambiar de directorio o carpeta. Por ejemplo, para ir a una carpeta específica de la unidad actual, se copia la dirección donde se encuentra esta, y se pega después de cd. Asimismo, para acceder a una subcarpeta dentro de una carpeta, simplemente se escribe “cd” seguido del nombre de la subcarpeta.



- **dir**: permite mostrar las carpetas y/o archivos contenidos en el directorio. Continuando con el ejemplo anterior, el comando se ejecutaría de la siguiente manera:

```
D:\Experimentos>dir
```

- **cls**: permite limpiar la ventana del cmd

```
D:\Experimentos>cls
```

- **mkdir**: permite crear un directorio o subdirectorio en la dirección actual. Si se desea crear una carpeta llamada ’ejemplo1’, se ejecuta lo siguiente:

```
D:\Experimentos>mkdir ejemplo1
```

- **copy con**: permite crear archivo txt en el directorio actual. En caso se desee editar uno ya existente, se utiliza el mismo comando y se acepta sobreescribir sobre el archivo. Para regresar al directorio actual, luego de terminar de editar el archivo, se presiona ctrl + Z, y después se presiona enter. A modo de ejemplo, si se decide crear un archivo txt de nombre "holamundo", se ejecuta lo siguiente:

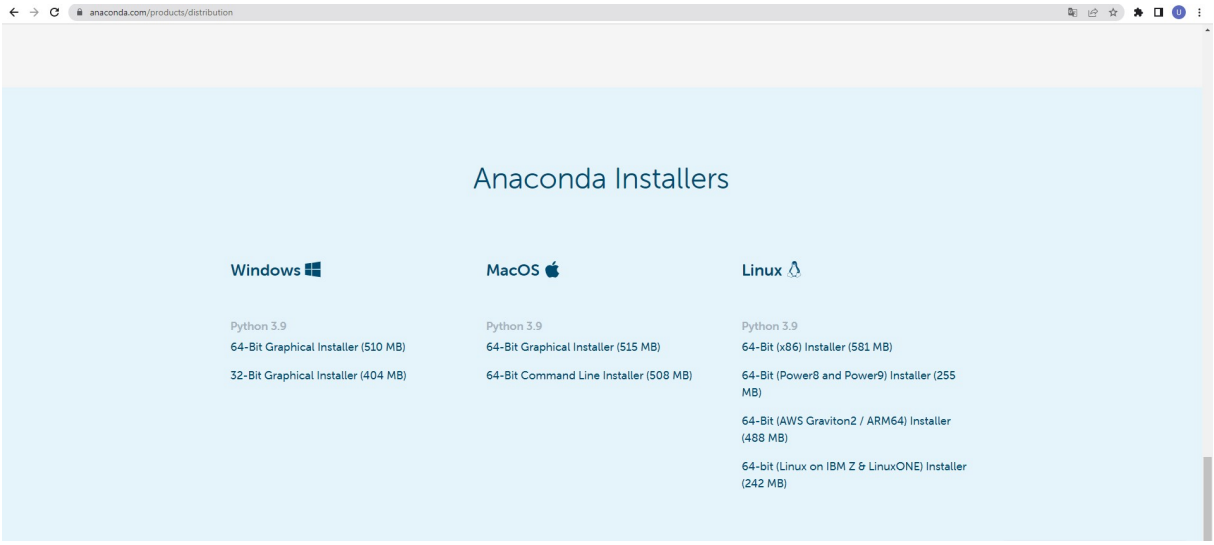
```
D:\Experimentos>copy con holamundo.txt
```

A.2. El Kit Esencial: Anaconda

En este apéndice se describe la instalación de Anaconda, y se detalla algunas funcionalidades importantes del Anaconda Prompt. Esto es útil, entre otras cosas, para instalar el paquete oTree tal, como se ve en el capítulo 3.

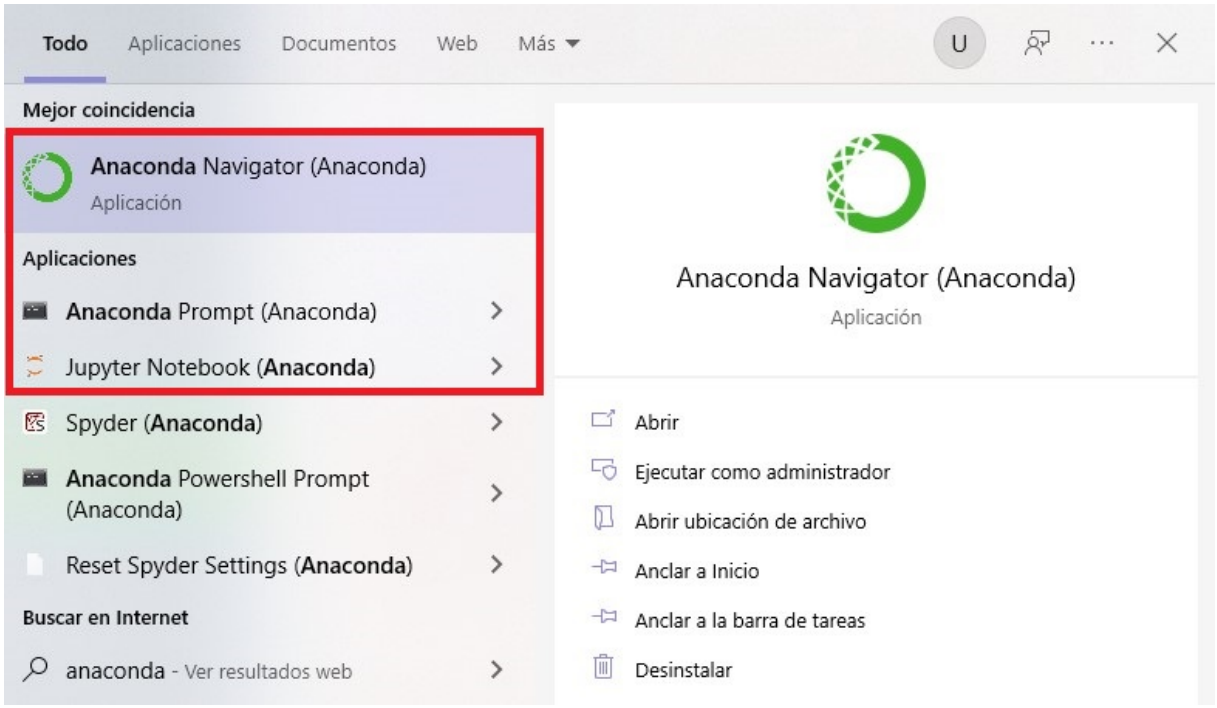
Instalación de Anaconda

Anaconda es un administrador de paquetes de código abierto de los lenguajes Python y R. Anaconda contiene Conda, el cual es un administrador de paquetes y environments, que trabaja con línea de comandos. En caso se quiera trabajar con una interfaz gráfica de usuario (GUI), se tiene Anaconda Navigator[1]. Lo primero es descargar Anaconda desde su página web[2]. Aparecerán las siguientes opciones:



Se elige la opción que corresponda al usuario. En este documento se usará Windows. Se instala Anaconda haciendo doble click en el archivo descargado, eligiendo las opciones predeterminadas. De esta manera, se tiene instalado Conda y Anaconda Navigator, así como Python y algunos paquetes. Para acceder a las aplicaciones de Anaconda, escriba en el buscador de su sistema operativo Anaconda, y le aparecerá una vista similar a la imagen de abajo. Las aplicaciones que se utilizan en este documento son Anaconda Navigator, Anaconda Prompt y Jupyter. A continuación de detalla el uso del Anaconda Prompt.

[1]Documentación de Anaconda: https://docs.anaconda.com/anaconda/
[2]Página de descarga de Anaconda: https://www.anaconda.com/products/distribution



Anaconda Prompt

La interfaz de Anaconda Prompt es muy parecida al CMD de Windows. A lo largo de este documento, se asume familiaridad con el CMD de Windows, por lo que se recomienda revisar el Apéndice A.1.

En base a la documentación[3], a continuación se describe el uso de Anaconda Prompt para el manejo de environments y paquetes,

- **Manejo de environments**: los environments son directorios que contienen una colección específica de paquetes Conda de manera aislada. Esto hace posible navegar entre diferentes environments sin dificultades. Cuando trabajamos en

[3]Documentación environments: https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/environments.html#why-use-venv-based-virtual-environments

Conda, tenemos por predeterminado el environment llamado "(base)", como se visualiza en la imagen anterior. La utilidad de los environments ocurre, por ejemplo, cuando deseamos instalar una aplicación específica, la cual no es compatible con la versión de Python de nuestro environment "(base)". Por lo tanto, es necesario crear uno nuevo para alojar la aplicación con sus paquetes y módulos.

Por ejemplo, para crear un environment llamado "Otree5" con la versión 3.9 de Python, se ejecuta:

```
conda create -n Otree5 python=3.9
```

Cuando Anaconda pregunte si se desea proceder, se presiona 'y'. Para ver la lista de todos los environments, incluyendo el nuevo, se ejecuta:

```
conda info --envs
```

Para trabajar en uno de esos environments, se debe activar mediante el comando:

```
conda activate Otree5
```

Ahora ya no aparece "(base)" al inicio, sino "(Otree5)", lo que significa que este último environment es el que está activo.

Para conocer la versión de Python del environment, se ejecuta:

```
python --version
```

Para retornar al environment "(base)", se ejecuta:

```
conda activate
```

Una vez en el environment "(base)", si se requiere eliminar el environment creado "(Otree5)", ejecute:

```
conda remove --name Otree5 --all
```

Se elige la opción "yes" para eliminar el environment.

- **Manejo de paquetes**: desde Anaconda Prompt es posible instalar nuevos paquetes, así como revisar la lista de paquetes ya instalados. Para instalar uno nuevo, primero se verifica si está disponible en el repositorio de Anaconda.

Por ejemplo, si se requiere instalar el paquete llamado “beautifulsoup4”, se ejecuta:

```
conda search beautifulsoup4
```

Aparecerá una lista de paquetes disponibles con ese nombre y las versiones correspondientes. Una vez que se verifica que el paquete está disponible, se procede a instalarlo con el comando:

```
conda install beautifulsoup4
```

Finalmente, para ver la lista completa de paquetes instalados, se ejecuta:

```
conda list
```

A.3. Instalación de oTree versión 3

Los pre-requisitos y proceso de instalación son los mismos que en la versión 5, con la salvedad de algunos detalles. Por esta razón, se sugiere revisar previamente el Apéndice A.2. Antes de la instalación de oTree 3, se crea otro environment, que se se llama Otree3, por ejemplo. Cabe indicar que La versión antigua de oTree solo es compatible con versiones de Python anteriores a las 3.8.x, por lo que se descargará la siguiente versión de Python.

```
conda create --name Otree3 python=3.7.11
```

Luego, accedemos a ese enviroment con el siguiente comando:

```
conda activate Otree3
```

Por último, ya dentro del enviroment Otree3, descargamos la versión 3 con el siguiente comando.

```
pip3 install -U "otree<5"
```

Cabe señalar que existen algunas variaciones puntuales en los formatos de la nueva versión respecto a la anterior. Al respecto, la documentación oficial de oTree hace un listado que puede encontrar en el siguiente enlace: https://www.otree.org/newformat.html.

A.4. HTML tags

HTML tags

Veamos algunos *html tags* básicos que se suelen utilizar en templates de oTree.

- ... : Dentro de la etiqueta <*p*> se introduce un texto como párrafo.
- <hX>. . . </hX>, donde X = 1, 2, 3, 4: Estas son llamadas etiquetas de encabezado numeradas. Es decir te da diferentes tamaños de letra dependiendo del propósito (título, cuerpo, subtítulo, etc). La etiqueta <h1> representa el tamaño de letra más grande, mientras que <h6>es la letra más pequeña.
- ... : la etiqueta se utiliza para poner en cursiva un texto.
- ... : la etiqueta se utiliza para resaltar un texto en negrita.
- ... ... : Estas son etiquetas de lista. denota lista y dentro de ella se encuentran los elementos <li>.

En la siguiente imagen puede notar como se ve una plantilla con los *html tags* anteriores.

```
{{block title}} HTML TAGS {{endblock}}

{{block content}}

<p>Los textos encerrados aquí son párrafos de texto</p>

<p>
    Los siguientes códigos que utilizan el hx(x=1,2,3,4) son para definir tamaños de letras.
    El h1 es el tamaño más pequeño. Mientras que el h4 es el tamaño más grande.
    Asimismo, podemos definir formatos de texto como <em>cursiva</em> o <b>negrita</b>
</p>

<h4>Bienvenidos</h4>
<h3>Bienvenidos</h3>
<h2>Bienvenidos</h2>
<h1>Bienvenidos</h1>

<p>
    También, podemos escribir una lista con viñetas de la siguiente
    manera:
</p>

<ul>
    <li>Cada jugador puede extraer del recurso pesquero entre 1 (mínimo) a 10 (máximo) unidades.</li>
    <li>No hay comunicación entre jugadores.</li>
    <li>No hay ninguna regulación para la pesca</li>
    <li>Finalmente, cada jugador toma su decisión </li>
</ul>
```

Asimismo, puede realizar tablas con etiquetas html utilizando `<table>`, `<thead><tr>`, `<th><tbody><tr>`, `<td>`. También puede colocar alguna referencia web con `<a><a href = 'link' >colocar_nombre </a>`, así como también imágenes con `<img scr = 'link' width = '500px' />`. La siguiente imagen ilustra un ejemplo de cómo utilizar las anteriores etiquetas.

```
<p> Agreguemos, por último, el código para agregar tablas, referencias e imágenes.</p>

<p>
    <h3>Tabla Histórica</h3>
    <table>
        <thead><tr>
            <th>ID<br></th>
            <th>Nombre</th>
            <th>Edad</th>
        </tr></thead>
        <tbody><tr>
            <td>1</td>
            <td>Luis</td>
            <td>23</td>
            </tr>
            <tr>
            <td>2</td>
            <td>Lorena</td>
            <td>45</td>
        </tr></tbody>
    </table>
</p>

<h3>Referencias</h3>
<p>
    Para buscar contenido de tablas ir al siguiente link:
    <a> <a href = 'htpps://www.tablesgenerator.com/html_tables'>html_tables </a>
</p>

<h3>Imagen de gatito</h3>

<img src="https://i.imgur.com/gM5yeyS.jpg" width="500px" />

{{ endblock }}
```

El resultado desplegándolo en la página web es el siguiente:

HTML TAGS

Los textos encerrados aquí son párrafos de texto

Los siguientes códigos que utilizan el hx(x=1,2,3,4) son para definir tamaños de letras. El h1 es el tamaño más pequeño. Mientras que el h4 es el tamaño más grande. Asimismo, podemos definir formatos de texto como *cursiva* o **negrita**

<h6>Bienvenidos</h6>
<h5>Bienvenidos</h5>
<h4>Bienvenidos</h4>
<h1>Bienvenidos</h1>

También, podemos escribir una lista con viñetas de la siguiente manera:

- Cada jugador puede extraer del recurso pesquero entre 1 (mínimo) a 10 (máximo) unidades.
- No hay comunicación entre jugadores.
- No hay ninguna regulación para la pesca
- Finalmente, cada jugador toma su decisión

Agreguemos, por último, el código para agregar tablas, referencias e imágenes.

Tabla Histórica

ID	Nombre	Edad
1	Luis	23
2	Lorena	45

Referencias

Para buscar contenido de tablas ir al siguiente link: html_tables

Imagen de gatito

Bibliografía

Bibliografía

[1] Amano, R., Kryvtsov, O., y Petersen, L. (2014). Recent Developments in Experimental Macroeconomics. Bank of Canada Review.

[2] Banerjee, A., Cole, S., Duflo, E., y Linden, L. (2007). Remedying Education: Evidence From Two Randomized Experiments In India. *The Quarterly Journal of Economics*, 122(3), 1235–1264. https://doi.org/10.1162/qjec.122.3.1235

[3] Bao, T., Hommes, C., y Pei J. (2021). Expectation formation in finance and macroeconomics: A review of new experimental evidence. *Journal of Behavioral and Experimental Finance*, 32.

[4] Barron, K., y Nurminen, T. (2020). Nudging cooperation in public goods provision.*Journal of Behavioral and Experimental* Economics, 88, 101542. https://doi.org/10.1016/j.socec.2020.101542

[5] Brañas Garza, P. (2022). *Economía experimental y del comportamiento.* Antoni Bosch.

[6] Card, D., y Krueger, A. B. (1993). *Minimum Wages and Employment: A Case Study of the Fast Food Industry in New Jersey and Pennsylvania* (No. 4509).

[7] Chen, D.L., Schonger, M. y Wickens, C.(2016). oTree - An open-source platform for laboratory, online and field experiments. *Journal of Behavioral and Experimental Finance*, 9, 88-97.

[8] Friedman, D., y Cassar, A. (2004). *Economics Lab: An intensive course in experimental economics*. Routledge.

[9] Galarza, F. y Power, M. (2012) Economía Experimental: nuevas metodologías para analizar el comportamiento individual. *Repositorio de la Universidad del Pacífico-UP*. Universidad del Pacífico. https://repositorio.up.edu.pe/handle/11354/2680

[10] Harrison, G. y List, J.(2004). Field Experiments. *Journal of Economic Literature*, 42, 1009-1055.

[11] Ifcher, J., y Zarghamee, H. (2015). Ethics and Experimental Economics. *Practicing Professional Ethics in Economics and Public Policy*, 195–205.

[12] Isaac, R. M., y Walker, J. M. (1988). Group Size Effects in Public Goods Provision: The Voluntary Contributions Mechanism. *The Quarterly Journal of Economics*, 103(1), 179. https://doi.org/10.2307/1882648

[13] Jacquemet, N. y L'Haridon, O. (2018). *Experimental economics: method and applications*. Cambridge University Press.

[14] Kahneman, D., y Tversky, A. (1979). Prospect Theory: An Analysis of Decision under Risk. *Econometrica*, 47(2), 263–292. https://about.jstor.org/terms

[15] Kim, O., Walker, M. The free rider problem: Experimental evidence. Public Choice 43, 3–24 (1984). https://doi.org/10.1007/BF00137902

[16] López, K., y Lugon, A. (2020). *Experimentos clásicos de economía. Evidencia de laboratorio de Perú (Documento de Trabajo)*. Pontificia Universidad Católica del Perú. https://doi.org/10.18800/2079-8474.0488

[17] Marwell, G., y Ames, R. E. (1979). Experiments on the Provision of Public Goods. I. Resources, Interest, Group Size, and the Free-Rider Problem. *American Journal of Sociology*, 84(6), 1335–1360. http://www.jstor.org/stable/2777895

[18] Moinas, S., y Pouget, S. (2013). The Bubble Game: An Experimental Study of Speculation. *Econometrica*, 81(4), 1507–1539. https://doi.org/10.3982/ecta9433

[19] Roth, A., y Kagel, J. (1995). *The Handbook of Experimental Economics*. Princeton University Press.

[20] Shefrin, H., y Statman, M. (2000). Behavioral Portfolio Theory. The Journal of Financial and Quantitative Analysis, 35(2), 127-151.

[21] Smith, V. L. (1962). An Experimental Study of Competitive Market Behavior. *Journal of Political Economy*, 70(2), 111-137. https://about.jstor.org/terms

[22] Smith, V. L. (1976). Experimental Economics: Induced Value Theory. *The American Economic Review*, 66(2), 274–279. https://about.jstor.org/terms

[23] Smith, V. L. (1994). Economics in the Laboratory. *The Journal of Economic Perspectives*, 8(1), 113-131.

[24] Thaler, R.H (1980) Toward a Positive Theory of Consumer Choice. *Journal of Economic Behavior and Organization*, 1, 39-60.

[25] Thaler, R. H., Tversky, A., Kahneman, D., y Schwartz, A. (1997). The Effect of Myopia and Loss Aversion on Risk Taking: An Experimental Test. *The Quarterly Journal of Economics*, 112(2), 647–661. https://www.jstor.org/stable/2951249

[26] Vicerrectorado de Investigación, Oficina de Ética de la Investigación e Integridad Científica. (2017). Módulo 1: ¿En qué consiste la ética de la investigación con seres humanos?. https://repositorio.pucp.edu.pe/index/handle/123456789/71120